Accelerated DPDK in containers for networking nodes

Rohan Krishnakumar

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 05.01.2019

Supervisor

Prof. Yu Xiao

Advisors

D.Sc.(tech) Vesa Hirvisalo

MSc. Kati Ilvonen

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Rohan Krishnakumar

Title Accelerated DPDK in containers for networking nodes

Degree programme Electronics and electrical engineering

Major Communications Engineering **Code of major** ELEC0007

Supervisor Prof. Yu Xiao

Advisors D.Sc.(tech) Vesa Hirvisalo, MSc. Kati Ilvonen

Date 05.01.2019 **Number of pages** 54+11 **Language** English

Abstract

NFV is fast moving towards containers for software virtualization and thereby imposing stringent requirements on container networking. The current status of networking in containers is quite limited especially for fast packet processing and needs further exploration.

The thesis is a comprehensive study on container networking for fast packet processing in standalone and orchestrated environments. Existing technologies such as DPDK and SR-IOV are utilized along with OvS and OVN to build different topologies and measure network latency. Utilizing these results and taking into account the hardware features supported by different nodes in an orchestrated environment, the thesis further proposes a latency based scheduler that can meet the network latency requirements of container applications being deployed.

Keywords Containers, CNI, DPDK, Docker, Kubernetes, NFV, OvS, OVN, SR-IOV

Preface

I would like to thank my supervisor Prof. Yu Xiao for her valuable guidance and insights throughout the course of this thesis. I would also like to thank my advisor Vesa Hirvisalo for his invaluable guidance and support during the course of my degree.

This thesis was conducted at Oy L M Ericsson Ab. I would like to thank my manager and advisor Kati Ilvonen and my team members for their constant encouragement. I would specifically like to thank Dietmar Fiedler, for suggesting the topic and reviewing the work.

Finally, I would like to thank my family and friends for supporting my studies.

Otaniemi, 04.12.2018

Rohan Krishnakumar

Contents

Abbreviations

API	Application Programming Interface
CNI	Container Networking Interface
DHCP	Dynamic Host Configuration Protocol
DMA	Direct Memory Access
DNS	Domain Name System
DPDK	Data Plane Development Kit
DUT	Device Under Test
EAL	Environment Abstraction Layer
GDB	GNU Debugger
HA	High Availability
IOMMU	I/O Memory Management Unit
IPC	Inter Process Communication
ISR	Interrupt Service Routine
KVM	Kernel-based Virtual Machine
k8s	Kubernetes
MAAS	Metal As A Service
NAT	Network Address Translation
NFV	Network Function Virtualization
NIC	Network Interface Card
OVN	Open Virtual Network
OvS	Open vSwitch
PCI	Peripheral Component Interconnect
PID	Process Identifier
POSIX	Portable Operating System Interface
RBAC	Role Based Access Control
SR-IOV	Single Root IO Virtualization
TLB	Translation Look-aside Buffer
VM	Virtual Machine
VMDQ	Virtual Machine Device Queue
VNF	Virtual Network Function
VT-d	Virtualization Technology for Directed I/O

1 Introduction

In recent years, the telecommunication industry has evolved towards Network Function Virtualization (NFV) where the traditional "hardware heavy" network node functions are being replaced by virtualized software that can run on generic hardware. Current de facto implementations of software virtualization utilize Virtual Machines (VMs) and is rapidly moving towards lightweight containers[11]. NFV is still in its nascent stages in understanding and utilizing containers and this thesis explores the suitability of containers in NFV from a networking perspective.

1.1 Thesis Scope and Objective

The scope of the thesis is a comprehensive understanding on the networking aspect of utilizing containers in NFV. Containers and container based deployments have so far found their use cases predominantly by replacing existing monolithic applications or by addressing environment specific requirements[22]. For instance, LinkedIn replaced their monolithic application with microservices in 2011 which resulted in less complex software and shorter development cycles[21]. Others such as NASA, have used containers to meet their dependency on specific versions of software, kernel or tool chain by packing all dependencies together in a container[46]. Utilizing containers in NFV is still in nascent stages and as a result, the current container networking is quite limited for NFV applications. Specifically, container run-time such as Docker and orchestrator like Kubernetes by default have just one interface attached to the container. Moreover, this interface is often used for management functions within the cluster. Exposing multiple interfaces and utilizing fast packet processing technologies on them is an important aspect in meeting the stringent networking requirements in NFV. Moreover, since most practical use cases of container applications utilize an orchestrated environment[16], such scenarios are also considered with equal importance. In short, the thesis tries to address these issues by exploring different networking scenarios and utilizing existing fast packet processing tools such as Data Plane Development Kit (DPDK) and Single Root I/O Virtualization (SR-IOV).

The objective of the thesis is to find a fast packet processing solution in a container orchestrated environment that meets the latency requirements of the deployed application container. The current container orchestrators such as Kubernetes and Apache Mesos do not consider network latency as an attribute while scheduling container applications. Application that are not latency sensitive such as a configuration management interface could consume nodes that provide best latency results in the cluster. The thesis considers different networking scenarios based on the availability of nodes and hardware features supported by each of them and proposes an intelligent way to deploy containers in the host that meets the latency requirement.

1.2 Thesis Contribution

The contribution of the thesis is a study on different fast packet processing solutions in containers in a standalone and orchestrated environment. To this extent, the author of the thesis has built a Kubernetes cluster on bare metal using Metal As A Service (MAAS), a software from Canonical[53] and kubeadm[56], a tool for bootstrapping Kubernetes. The author of the thesis has measured network latency for different networking scenarios utilizing primarily DPDK, SR-IOV, Open vSwitch (OvS) and Open Virtual Network (OVN). Various Container Networking Interface (CNI) plug-ins were utilized to realize multiple interfaces in containers and use the above technologies. The author of the thesis has taken the latency measurement values as a benchmark and has implemented a Kubernetes scheduler that utilizes these values to intelligently schedule container applications considering their network latency requirements.

1.3 Thesis Structure

The next chapter in the thesis is the background which covers the related tools and technologies utilized, their definitions and how they work. It also covers their motivation and in some cases their evolution. The third chapter covers related work in the field of interest and how they compare to the work carried out in the thesis. The fourth and fifth chapters are the core part of the thesis. The motivation for the experiments, the test topologies and the various measurements and exploration of different networking scenarios are covered in the fourth chapter. The fifth chapter utilizes the measurements from chapter four as a benchmark and addresses the need for a scheduler that considers the latency requirement of the deployed container application in an orchestrated environment. The final chapter concludes the thesis and indicates some of the future work that could be carried out.

2 Background

In telecommunication industry, the network node functions have traditionally run on operator proprietary hardware which are dedicated for a specific purpose. This has led to long development and deployment cycles, higher cost of operation, etc. thereby hindering rapid technological advancement in the field. Slowly the network node functions are being replaced by virtualized software solutions that can run on generic computing hardware, consequently addressing the above problems and making the network more flexible. This network architecture where virtualized software solutions are chained together to form network node functions is called Network Function Virtualization. In the NFV nomenclature, the software implementation of the network node functions or sub-functions are referred to as Virtual Network Functions (VNF) - for instance, a network firewall, a DHCP server or a NAT function. The current de facto VNF implementations run on virtual machines such as Kernel-based Virtual Machine (KVM). But with the recent advent of containers and their gaining popularity, virtual machines are being replaced by containers in virtualized software. Containers are comparatively lightweight and utilize the same host kernel when compared to virtual machines. They can be booted up with minimum latency[13] implying that container based deployments focus on on-demand spawning of the containerized application. For example, a NAT application running on a container could be spawned up based on the number of clients utilizing it, with one container handling one or a fixed number of clients.

As depicted in Figure 1, a container orchestrated environment could be visualized as a cluster of multiple hosts with containers being spawned on request and killed after serving the request. The orchestrator runs on one host and acts as the master. It normally has a scheduler that decides on which of the other hosts to deploy the container, based on the provided requirements of the container, current status of the cluster and different hardware features supported by different hosts. In the scope of the thesis, the hardware features that are of interest are the availability of DPDK and SR-IOV interfaces. Depending on which node the container application is deployed, the latency could be affected significantly. There are three different scenarios considered in depth in Chapter 4.3 which also motivate the need for a scheduler that is aware of the latency requirements of the application and the capabilities of the system.

The rest of the chapter is divided into subsections that cover the tools and technologies that are integral to the thesis. The first two subsections cover DPDK and SR-IOV which are fast packet processing technologies. The third subsection gives an in-depth understanding on containers and their development from concepts such as namespaces and control groups. The container orchestration system, specifically Kubernetes is discussed in the fourth subsection. The final subsection briefly covers OvS.

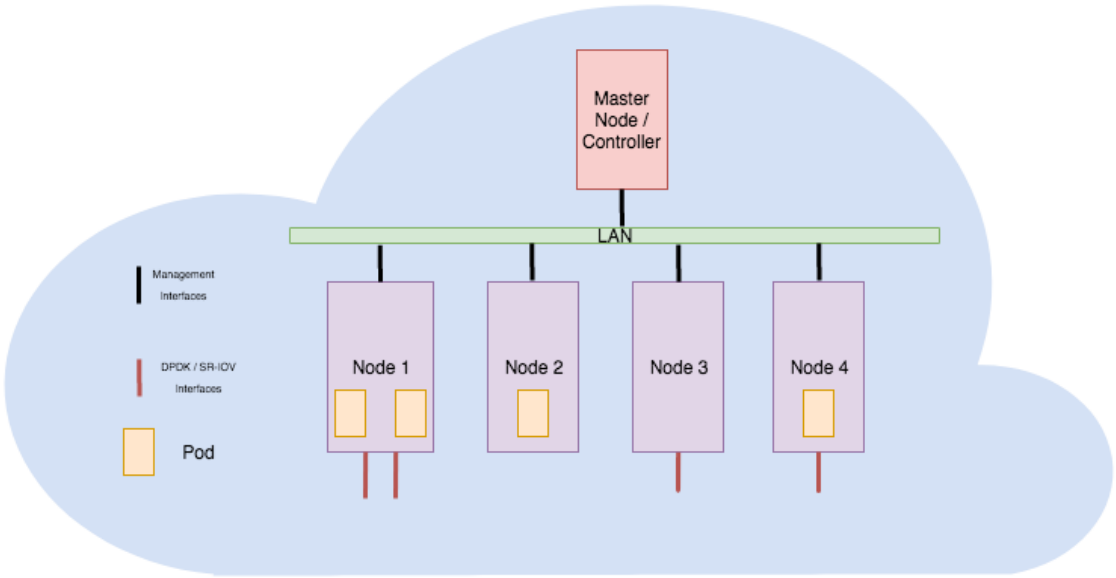

Figure 1: A cluster of nodes with one acting as a master. Container applications or pods run on other nodes that have different hardware features.

2.1 Data Plane Development Kit, DPDK

DPDK is a "set of libraries and drivers"[25] that enable fast packet processing. It is an open-source software framework that can be used to build fast networking applications and is maintained by the Linux Foundation[26]. As compared to a normal kernel network stack, DPDK can provide about twenty five percent improvement in packet processing speed.

The driving factor behind the development of DPDK was to address the handling of extremely fast packet rates, especially in a communication network infrastructure, where the packet sizes are typically smaller and packet rates much higher. The average CPU cycles available for handling one packet in such cases are quite small. For instance, a 10 Gigabit Ethernet card receiving packets of size 1024 bytes could potentially receive 1.25 million packets per second. A CPU with 2GHz clock cycles handling these packets would have on average 1600 cycles per packet. But when the packet size decreases to, say 64 bytes, these values could be 19.5 million packets per second and 102 CPU cycles available per packet which is too small a value when using kernel network stack.

DPDK utilizes a variety of techniques to improve packet processing. Primarily, it uses Poll Mode Drivers (PMD) instead of interrupt driven ones for packet handling. As shown in Figure 2, in the traditional network stack, when a packet is received, the CPU does a context switching from the user space process to kernel. It then runs the Interrupt Service Routine (ISR) and switches back to user space. This is a big overhead especially for higher packet rates. Poll Mode Drivers run on the user space and use a dedicated CPU core to handle the traffic on one or multiple interfaces.

The CPU continuously polls the kernel network driver and the utilization hits close to hundred percent. This can be checked with `top` command in Linux. DPDK uses POSIX thread affinity to disable the kernel scheduler from utilizing these cores for other processing.

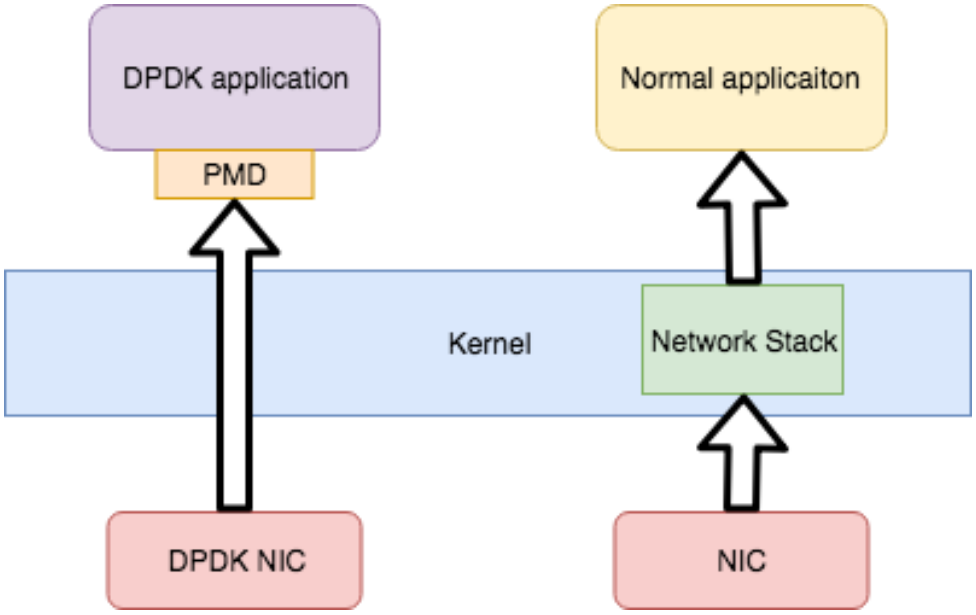

Figure 2: Packet processing comparison between DPDK based application and via the Kernel

DPDK uses Hugepages to improve performance in packet processing. Hugepages refer to pages in main memory with bigger page size. The standard Linux page size is 4kB while with Hugepages, the size could be 2MB or even 1GB. For some specific use cases, even higher values are used. During a normal memory access when a process is handled by the CPU, the virtual memory address used in the program needs to be translated to physical memory address before it can be accessed from the main memory. In the different memory access stages, the critical factor affecting performance is the Translation Look-aside Buffer (TLB) hit. When bigger page sizes are used, there are fewer pages and higher probability of a TLB hit[5]. However, Hugepages are optimal only in scenarios such as network packet handling where processes utilize enough memory to consume the allocated page size.

Another feature utilized in DPDK is Direct Memory Access (DMA). DMA is used to speed up copying of packet buffers. Compared to a normal kernel network stack, when a packet is received, DPDK uses DMA to directly lift the buffer to user space. The traditional packet handling involves copying of buffer multiple times such as, from the Network Interface Card (NIC) buffer to kernel socket buffer (`skbuf` in Linux) and from there to the user space. This reduces performance due to the copying overhead and also due to loss of localization of data leading to fewer cache hits. The method used in DPDK is called zero-copy. Moreover, DPDK uses cache alignment. In Intel machines, cache lines are normally 64 bytes. If the data fetched

from memory is placed in different cache lines, it would require multiple fetches from cache to retrieve the entire data when compared to data being present on the same cache line.

There are two modes of operation of a DPDK application, run to completion and pipeline. In run to completion, each core is assigned a port or a set of ports. The CPU core that is assigned to the port does the I/O specifically for that set of ports, accepts the incoming packets, processes and sends it out through the same set of ports. On the other hand, in pipeline mode, one core is dedicated to just I/O and processing is handled by other cores. All packets received at a port are sent to different cores using ring buffers. This method is more complex and could result in packet order not being maintained.

The main components of DPDK are the core libraries, Poll Mode Drivers for various supported interface cards and other libraries that deal with packet classification, Quality of Service (QoS), etc.[24]. All of these libraries run on the userpace. Apart from theses, DPDK uses Peripheral Component Interconnect (PCI) drivers in the kernel such as IGB-UIO or VFIO-PCI that perform the basic PCI functionality. When an interface is bound to DPDK, it is detached from the normal kernel driver and attached to one of these DPDK drivers in the kernel.

The core libraries in DPDK have an Environment Abstraction Layer (EAL) which hides or abstracts the platform from the libraries and applications running above it. It also handles memory allocation in Hugepages and PCI related buffer handling. DPDK does not allocate memory at run-time but rather pre-allocates memory during initialization of the application. For packet processing, it uses memory from this pool and returns it back to the pool after use. The library that allocates the pool of memory (Mempool) consists of memory buffer (Mbuf) and lockless queues or ring buffers for handling them, and are all part of the set of core DPDK libraries.

2.2 Single Root IO Virtualization, SR-IOV

SR-IOV is the hardware virtualization of a PCI Express (PCIe) device, or specifically, a Network Interface Card, into multiple virtual devices that can be directly assigned to an instance of a virtual machine or a container. SR-IOV has a Physical Function (PF) which acts like a normal PCIe device and multiple Virtual Functions (VF) that are lightweight functions with dedicated Rx/Tx queues[17]. In a virtual machine context, the host machine kernel can be bypassed and the packets can flow directly between the VF and PF.

Consider the scenario where one host is running multiple virtual machines (VMs) and a network interface card (NIC) receiving and transmitting packets for all those VMs. Without any hardware virtualization, when a packet is received by the NIC, it triggers an interrupt to the CPU core that is assigned to handle NIC interrupts. This core services the request and examines the packet. Based on the MAC address or VLAN tag, it forwards the packet to the correct VM by triggering another interrupt on the core that is servicing the virtual machine. This is an overhead since there are multiple interrupt handling in the host even before the packet is transferred to the guest operating system. There is also an extra copy of the packet buffer from

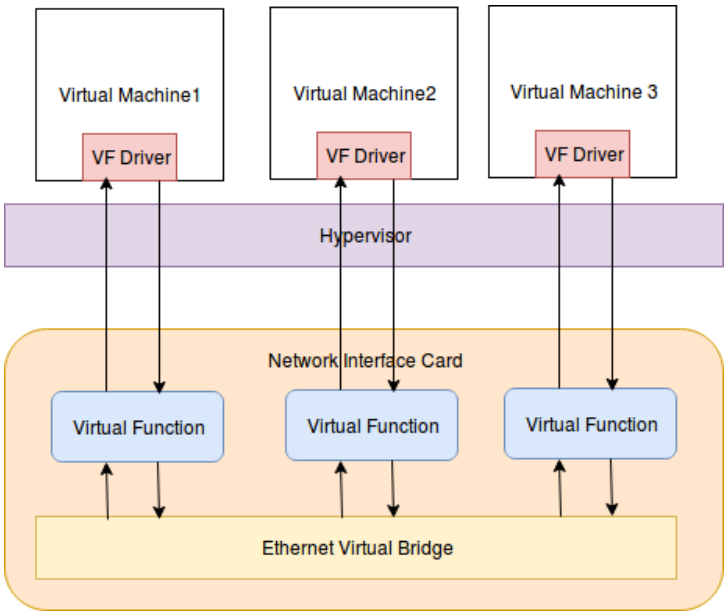

Figure 3: SR-IOV interface with Ethernet Virtual Bridge that sorts packets into one of the Virtual Functions and forwards to the Virtual Machine

the host to the VM. Moreover, when the packet rate is high, one core handling all the packets for all the VMs can be a severe bottleneck. As an solution to this, Intel proposed Virtual Machine Device Queues (VMDQ) [18] technology which has separate packet queues for each core. The received packets are put into one of the queues based on the destination MAC or VLAN. Each core services its packet queue by copying the buffer to the VM. There is distribution of work here and a good performance enhancement. An even further enhancement is SR-IOV in which the NIC has separate packet queues for each Virtual Function. When a packet is received, it is sorted to one of the VF queues based on the destination MAC address or VLAN tag by a virtual bridge or classifier. The VF then pushes the packet up the virtual machine directly without involving the CPU using Direct Memory Access (DMA). SR-IOV needs CPU I/O virtualization technology such as Intel VT-d (Virtualization Technology for Directed I/O) or AMD-Vi (AMD I/O Virtualization) enabled for the DMA to the virtual machine[20]. Figure 3 depicts the functioning of Virtual Functions in SR-IOV.

2.3 Container

A container is an isolated execution environment which includes any necessary run time dependencies and libraries needed for the application that it packages. It could be thought of as a software that runs on the host with its own slice of the system resources such as network, file systems, user groups, etc. but isolated from other processes. Containers are built on namespaces and control groups which are a part of the Linux kernel.

Namespaces

As per Linux man pages, namespace is an abstraction layer on "global system resources" that provides isolated instance of the resource to the processes that are members of the namespace. Linux provides six different namespaces, Inter Process Communication (IPC) including POSIX and System V message queues, Network, Mount, Process Identifier (PID), User and Unix Timesharing System (UTS)[27].

With network namespace, the processes in the new namespace can add new network devices without they being available to host network namespace and vice versa. Similarly, with mount namespace, different file systems can be mounted and processes within this namespace cannot access other host file system. Namespace can be created using Linux system call `unshare`[29]. The context of namespace that every process belongs to, is stored in a sub-folder under proc file system as shown below.

```
$ ls -l /proc/$$/ns | awk '{print $1"\t"$(NF-2)$(NF-1)$NF}'
lrwxrwxrwx      cgroup->cgroup:[4026531835]
lrwxrwxrwx      ipc->ipc:[4026531839]
lrwxrwxrwx      mnt->mnt:[4026531840]
lrwxrwxrwx      net->net:[4026532009]
lrwxrwxrwx      pid->pid:[4026531836]
lrwxrwxrwx      pid_for_children->pid:[4026531836]
lrwxrwxrwx      user->user:[4026531837]
lrwxrwxrwx      uts->uts:[4026531838]
```

Below is an example of using namespace to create an isolated environment. The current process identifier in a bash shell can be found out using `$$`. Then use `unshare` command to create a new namespace. Here the arguments "pid" and "net" signifies new namespaces for process identifier and network. The argument "mount-proc" tells the command to mount the proc file system. The "fork" option makes a fork of the current process before creating the namespace and attaches the child process to it[29].

```
$ echo $$
8009
$ unshare --fork --pid --net --mount-proc bash
root:~# ip l
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode
   DEFAULT group
    default qlen 1000 link/loopback 00:00:00:00:00:00
    brd 00:00:00:00:00:00
root:~# ps -aef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 13:37 pts/26   00:00:00 bash
root        12     0  0 13:44 pts/30   00:00:00 -bash
root        32     1  0 13:52 pts/26   00:00:00 ps -aef
root:~#
```

The new namespace created for the container is evident from `ip link` command as it does not list any of the interfaces available on the host machine. The command `nsenter`[28] can be used to enter a namespace as shown below. The argument `-t` takes process identifier of the target process and joins its namespace. The other arguments correspond to the different namespaces of the target process that it should enter.

```
$ ps -aef | grep bash | grep root | awk '{print $1"\t"$2"\t"
   $3"\t"$8}'
root      31215  8009    sudo
root      31294 31215    sudo
root      31295 31294    unshare
root      31296 31295    bash

$ nsenter -m -u -i -n -p  -t 31296
root:/# ps -aef
UID         PID  PPID  C STIME TTY          TIME CMD
root          1     0  0 13:37 pts/26   00:00:00 bash
root         12     0  0 13:44 pts/30   00:00:00 -bash
root         30    12  0 13:45 pts/30   00:00:00 ps -aef
root:/#
```

As a final step, add a network bridge within the new namespace and it is possible to verify that this bridge is not available in the host namespace.

```
root:~# brctl addbr asd
root:~# ip l
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode
   DEFAULT group
    default qlen 1000 link/loopback 00:00:00:00:00:00
    brd 00:00:00:00:00:00
2: asd: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN
   mode
    DEFAULT group default qlen 1000 link/ether 4a:18:f1:3c:9c
       :b2
    brd ff:ff:ff:ff:ff:ff
```

Control Groups

Control groups on the other hand are useful in allocating and limiting resource consumption such as memory, CPU time, network bandwidth etc. Control groups or cgroups are organized hierarchically, child cgroups inherit from parent cgroups. The pseudo file system "sys" is used as an interface to create and manipulate cgroups. To create a new cgroup, create a directory under `/sys/fs/cgroup` and set appropriate values. For instance, when a Kubernetes pod is created with Hugepage limited to 4GB as described here, cgroups creates a directory in `/sys/fs/cgroup/hugetlb/`

`kubepods/` and sets value of `hugetlb.1GB.limit_in_bytes` to 4294967296 in bytes which is equivalent to 4GB.

```
$ sudo cat /sys/fs/cgroup/hugetlb/kubepods/hugetlb.1GB.
   limit_in_bytes
4294967296
```

Comparison to Virtual Machines

Any description of containers is incomplete without a comparison to virtual machines. As depicted in Figure 4, a virtual machine emulates an entire machine. There is a hypervisor that runs on the host machine that emulates the underlying hardware and a complete copy of the guest operating system running on top of it. The advantage of virtual machines is that, it is completely isolated from the host machine. The execution environment can be guaranteed in spite of the host hardware and operating system. While at the same time, virtual machines can be heavy weight and resource consuming. In contrast, a container essentially runs completely on the host. It uses namespaces and control groups for isolation of the container from the host. As a result, they are much more easy to boot up, lightweight and picking up in popularity when compared to virtual machines [13].

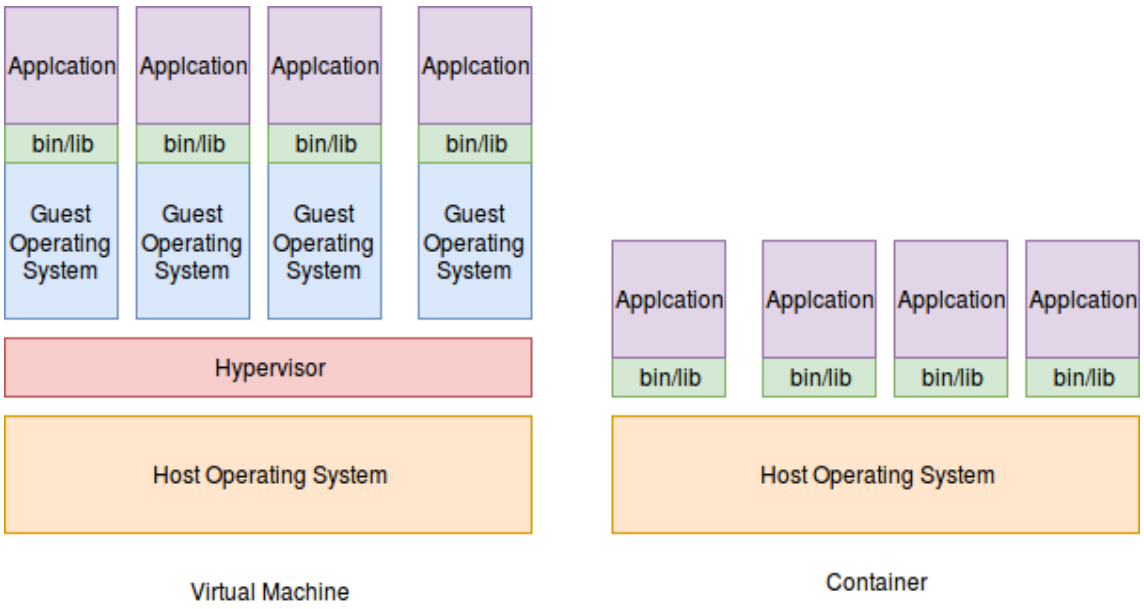

Figure 4: Comaprison between virtual machines and containers

2.3.1 Container Networking Interface

Container Networking Interface Specification is a proposal for standardized networking solution for container applications. It is documented in GitHub and has its origins from

CoreOS and rkt[31]. According to this specification, container run-time is responsible for creating the network namespace and the network plug-in is an executable that can be called by the container during execution. It specifies functions that should be exposed by the network plug-in such as ADD, DEL, GET and VERSION. For instance, ADD is invoked with the container identifier, network namespace and other network configurations. This triggers the plug-in executable to insert an interface into that namespace as per the configuration. Similarly, DEL is invoked when the container is killed.

2.4 Kubernetes

Kubernetes is an open source container orchestration system. It is used for deploying and managing container applications in a cluster. It is a result of evolution of container management system developed by Google internally, known as Borg. Borg was then replaced by Omega and then finally by Kubernetes[32]. It was made open source in 2014[33].

Typically a Kubernetes cluster consists of at least one master node and many worker nodes. The master node is responsible for maintaining the cluster state, managing the resources, scheduling the work load, etc., while the application containers actually run in the worker nodes. The smallest unit that can be controlled by Kubernetes is a pod. A pod can have multiple containers running within it but all of them share the same namespace. So, if there are multiple containers that need to interact closely, for instance, through inter process communication (IPC), they can be placed in the same pod, since the containers in the same pod will share the same IPC namespace.

The various functionalities of a Kubernetes master node are handled by different components such as Application Programming Interface (API) server, controller manager, etcd and scheduler. These components do not necessarily have to be run on one master node but could be run anywhere in the cluster. But as a good practice, to support High Availability (HA), it is encouraged to run these control plane components in one node[34].

The API server exposes API's that are used by other control plane components to retrieve, store and manage information about the cluster. It also acts as an endpoint for communications between the cluster and the Kubernetes master.

The Kubernetes scheduler is responsible for scheduling the workload between the worker nodes. Typically when a pod is requested to be deployed, the scheduler considers the resource requests in the pod and runs filtering and prioritizing functions and binds the pod to the node with the highest priority. The filtering run filters out nodes that cannot support the pod. For instance, as when a pod requests for Hugepages and the nodes that do not support Hugepages are filtered out. The priority function assigns a priority to all the nodes that passed the filtering step. Effectively, the priority function does some load balancing between the worker nodes considering primarily CPU and memory pressure on the nodes.

etcd is a database that stores Kubernetes cluster information. It stores key value pairs and the API server is the only component that interacts with it.

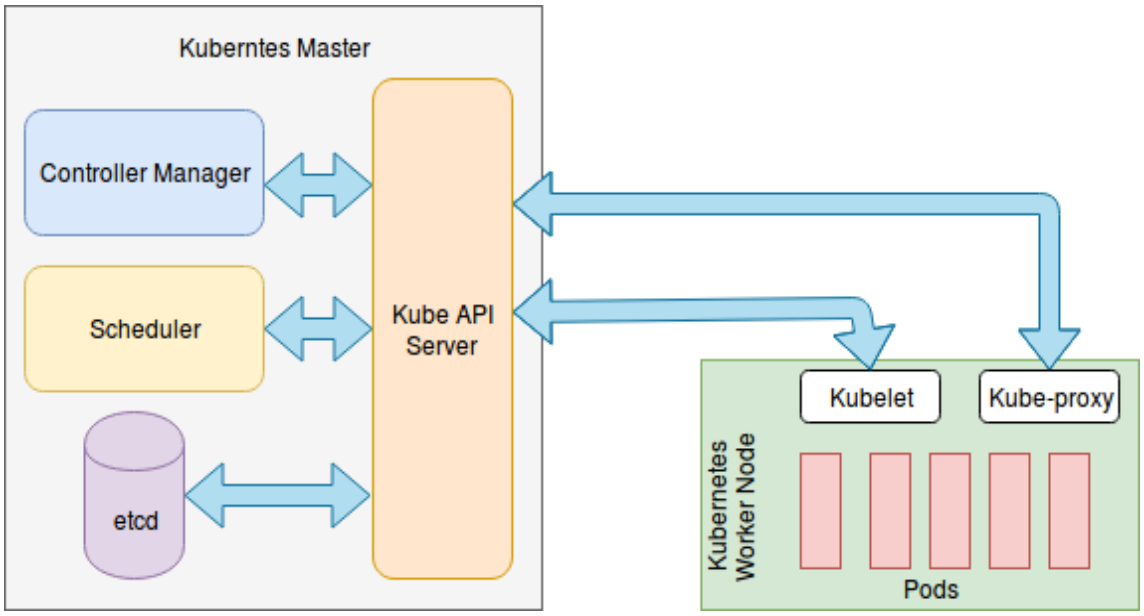

Figure 5: Various components Kubernetes master and worker nodes and the way they interact

Controllers in Kubernetes monitor the state of various resources and try to bring them to the desired state. For instance, the replication controller makes sure the desired number of pods are running in the cluster. If there are too many, the unnecessary ones are killed and if there are fewer, the required number of pods are created. The collection of all such controllers into one application is the controller manager. There is also a cloud controller manager in the Kubernetes master that interacts with the underlying cloud provider such as Google Compute Engine or Amazon Web Services.

Kubernetes worker nodes contain kubelet, kube-proxy and the container runtime, Docker. Kubelet manages the pods running in a node. It receives the pod configuration from the API server and deploys the pod. It also sends back logs and is used for connecting to a pod, for instance through the `kubectl exec` command.

Kube-proxy handles the networking part of the worker nodes. For instance, it implements iptable rules for port forwarding and exposing services running on the cluster[35].

`kubectl` is a command line tool that is used to manage and deploy Kubernetes applications. The easiest way to deploy a pod running Ubuntu image is given below.

```
$ kubectl run --image=ubuntu:latest test-pod /bin/bash
deployment "test-pod" created
$ kubectl get pods
NAME                          READY     STATUS    RESTARTS
   AGE
test-pod-79b4848dd4-w4srg     1/1       Running   0          8s
```

2.5 Open vSwitch

Open vSwitch is an open source implementation of a virtual switch which is designed for virtualized environments such as a data center. In essence, it could be considered as a "feature rich" Linux bridge[36].

The main components of OvS are the database server, the main process daemon or `ovs-vswitchd` and a data path which is normally a kernel module. Apart from these main components, OvS uses mainly three protocols for communicating between them, namely, OVSDB management protocol[38], OpenFlow[12] and Netlink[30]. The way these components interact along with management tools is depicted in Figure 6

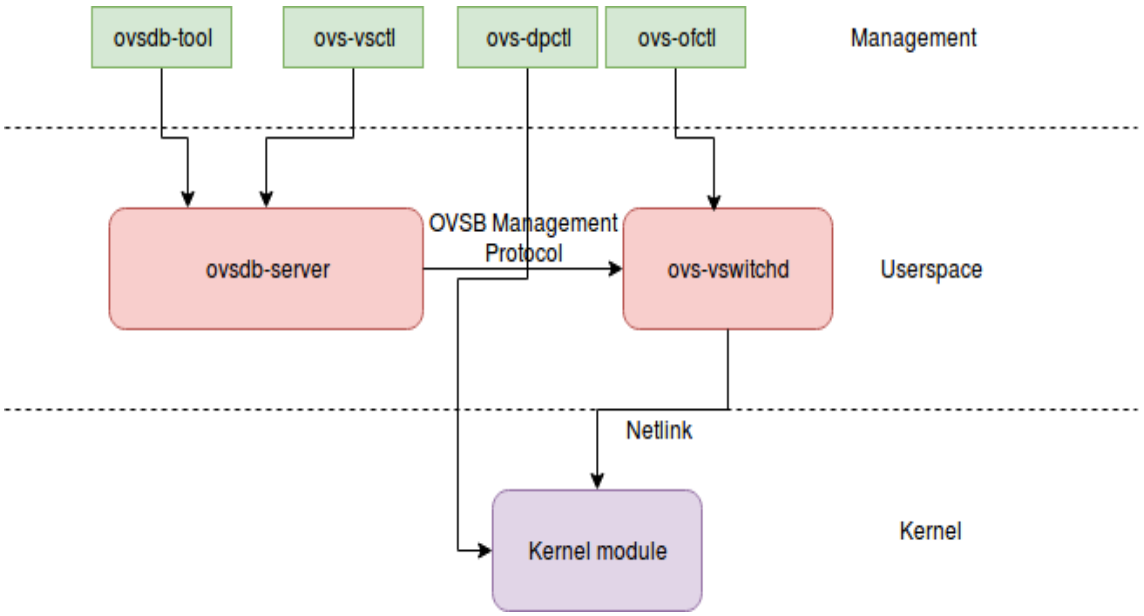

Figure 6: Various components in Open vSwitch separated into management layer, user space and kernel

The database server or `ovsdb-server` is a persistent storage of OvS configuration. It is configured using an external configuration tool such as the OvS CLI tool `ovsdb-tool` or `ovs-vsctl`. The former is primer for configuring the database while the latter is for configuring the switch. The database server communicates with `ovs-vswitchd` to exchange configuration information using OVSDB management protocol[38].

The main process daemon, `ovs-vswitchd`, is responsible for managing all the OvS bridges in the machine. Network flows are installed using an external controller such as `ovs-ofctl`. It then installs these flows in the data path or the kernel module using Netlink.

In this thesis, OvS is configured to run on DPDK. This implies that instead of the kernel module, the forwarding plane resides on the user space. The physical interfaces use Poll Mode Drivers (PMD) running in the user space.

3 Related work

There has been considerable interest in DPDK based packet processing and further improvements on it. Optimization of DPDK applications could be considered on a use case basis and there are quite a few parameters that can be tweaked for better performance such as the number of CPU cores used for Rx and Tx, Hugepage allocation and isolation, NUMA locality, buffer allocation, etc. Bl H. et al, explore further improvements on DPDK based packet processing in the paper "DPDK-based Improvement of Packet Forwarding" [1]. The focus is on DMA and zero copy methods that are used in DPDK. The paper proposes an improvement in the DMA method by reducing the DMA operation during a packet buffer copy to user space. Similarly, Cerrato I. et al, analyze multiple DPDK based applications running on the same host with OvS in "Supporting Fine-Grained Network Functions through Intel DPDK" [43]. In their work, OvS runs on primary CPU core and the network functions on other secondary ones. The packets are received at OvS and distributed to the correct network function and flow through a service chain and finally out of the host through OvS. The study here is optimizing performance by assigning different number of queues to handle input and output flows.

The research papers mentioned above primarily address optimizing the performance of DPDK based applications. This is not in the scope of this thesis and could be considered for future work. Improvements in packet processing in NFV is addressed by Kourtis et al, in "Enhancing VNF Performance by Exploiting SR-IOV and DPDK Packet Processing Acceleration" [7]. The paper considers SR-IOV and DPDK as the primary methods of fast packet processing and utilize them in a virtualized environment consisting of virtual machines. The study focuses on performance enhancement using these techniques and compares them to processing of packets using Linux kernel network stack. This is very similar to the experiment carried out in 4.1 in this thesis where, in contrast containers are used.

A common problem in virtualized environment is the interface between a hypervisor and the guest operating system to emulate physical NIC. Virtio is the industry standard used and is considered to be much less resource intensive that emulating a complete PCI device. The underlying concepts are explored by Tan J. et al, in [9] and Russell R., in [10]. The concepts understood from these research work such as, `vhost` and `virtio` queues are used to make a better judgment on the latency measurements in later section, 4.2.

Networking solution for containers are gaining in interest with the increasing popularity of containers. "FreeFlow" [3] is a networking solution in an orchestrated environment and argues that complete isolation between containers running similar applications is not necessary. It also use Remote Direct Memory Access (RDMA) for some performance gains. "An Analysis and Empirical Study of Container Networks" [16] considers situations where containers are spawned on demand and argues that establishing a dynamic network for such containers can add considerable delay in container boot-up time. The research also argues such network incurs throughput loss and adds latency. The work carried out in "Minimizing latency of Real-Time Container Cloud for Software Radio Access Networks" [51] is quite close to the scope

of the thesis. This explores container networking solution for low latency using primarily DPDK and a real time kernel. The real time kernel gives priority to latency sensitive processes even under heavy work load. OvS with DPDK is used with Kubernetes for inter node communication and for data packet forwarding. The work concludes that DPDK is essential for a latency sensitive solution. Some of the latency measurements stated in the paper are compared with the values obtained in the latency measurements conducted for the thesis.

Other miscellaneous work of interest include "Iron" [14] which argues that complete CPU resource allocation and isolation using control groups is generally overlooked. When a process or a container is handling fast network traffic, the time spent by the CPU handling hardware or software interrupts is unaccounted for. For software interrupts, the CPU time for interrupt handling is accounted in the process that was interrupted and not necessarily in the process that generated it. This can break CPU resource time division. Szabo et al, explore the idea of resource orchestration based on CPU locality in "Making the Data Plane Ready for NF" [15], which could also be extended to Kubernetes using Node Feature Discovery. "Scalable High Performance User Space Networking for Containers" [8] details some good testing topology using DPDK and mentions that Hugepage isolation in containers is not present. This is again an important requirement while running multiple containers in the same host and is noted for future work.

Most of the related work mentioned above focus specifically on some technology such DPDK, SR-IOV, virtio, etc., all of which are used in the thesis. However, their focus is primarily on optimizing these applications for better performance. Work conducted by Kourtis et al. in [7] and Mao et al. in [51] are the closest in scope to the thesis. As noted before, [7] uses Virtual Machines for NFV. Utilizing containers is a more recent field and uses different technologies for multiple interfaces and orchestration. Mao et al. on the other hand, utilize containers in stand alone and orchestrated environments. In the orchestrated environment, a Kubernetes cluster with OvS for an overlay network is used and experiments were conducted to measure network performance in both of these scenarios. In comparison, this thesis explores networking solutions in a broader scope using third-party CNI plug-ins which have become an important part of container ecosystem. More importantly, the thesis extends the network latency measurements and addresses the need for latency aware scheduling in an orchestrated environment. This consequently reduces the hardware requirement on the network nodes and retains the flexibility offered by container orchestration.

4 Latency Measurements

This chapter covers the experiments done for the thesis. There are different topologies considered essentially for containers in standalone and orchestrated environments. In all the topologies, the packet generator side is kept the same. The packet generator used for all the experiments, is a DPDK based application `pktgen` which can generate packets up to a rate of 14 million packets per second. Emmerich P. et al. do a comprehensive study on different packet generators considering throughput, precision and how they are affected at different CPU frequencies and architectures in [19]. In these experiments, DPDK based `pktgen` proved to be a good choice and is used in the thesis. The packet size used is 96 bytes to include time stamps in the buffer. The latency measurements are done at the packet generator side. This implies the time stamps are placed on the packets at the generator and compared with reception time after a complete round trip as represented in Figure 7 with green arrows. The packet generator and the Device Under Test (DUT) are connected through optical fibres using SR-IOV interfaces.

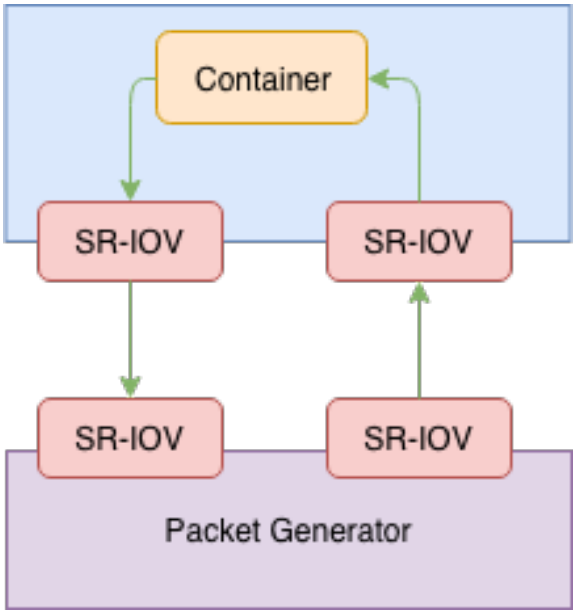

Figure 7: Test setup used for latency measurements. The flow of packets is depicted with green arrows.

Rest of the chapter is divided into subsections that detail the different experiments and the results obtained. The first two subsections explore a fast data path in container running on a single host. The first of those is a container running a DPDK application with interfaces directly exposed to the container. The second subsection utilizes OvS for forwarding packets within the DUT. There is also a comparative study to understand how much acceleration is brought about by DPDK.

The last two subsections consider only a cluster environment with Kubernetes. For a fast data path in this case, multiple scenarios have been considered and some

potential solutions discussed and experimented with. For this, initially a Kubernetes cluster is built on bare metal and then different scenarios are outlined and discussed one after the other in a detailed manner.

4.1 Container on DPDK and SR-IOV

A default docker container built on a bare Ubuntu image creates by default one interface that is connected to "docker0" bridge on the host. This is the default networking option in Docker [41]. The following illustrate running a Docker container with Ubuntu image and listing its interfaces.

```
$ docker run -it ubuntu:latest /bin/bash
root@f8344e977497:/# ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.0.2  netmask 255.255.0.0  broadcast
           172.17.255.255
        ether 02:42:ac:11:00:02  txqueuelen 0  (Ethernet)
        RX packets 25  bytes 4140 (4.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0
           collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0
           collisions 0
```

The bridge that is created automatically in the host, "docker0", resides in the same sub-net and acts as a gateway. Multiple Docker containers on the same host can communicate with each other using this bridge.

```
$ ip a | grep -A 3 docker0
7: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500
   qdisc noqueue state DOWN group default
    link/ether 02:42:29:09:76:49 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.1/16 brd 172.17.255.255 scope global
       docker0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:29ff:fe09:7649/64 scope link
       valid_lft forever preferred_lft forever
```

Docker supports other networking options such as host, overlay, macvlan [41]. Host networking option in Docker removes the network isolation between the host and the container. The overlay option is used with Docker swarm. It creates an overlay network and can be used to communicate with containers in other hosts. Macvlan assigns a MAC address to the virtual interface in the container and exposes it to a parent host physical interface making it look like a real physical interface. All of the above options are suited for specific use cases. The final option in Docker networking, using network plug-ins, is the most interesting one. It is an interface to use third part networking plug-ins, or Container Networking Interface (CNI) plug-ins. The open source CNI plug-ins for SR-IOV and DPDK could be used to explore fast data path networking in containers and is used for experiments below.

The test setup consists of two physical machines. Both the machines have a management interface and two SR-IOV interfaces. The management interface is used to configure them remotely using Secure Shell (SSH). The SR-IOV interfaces are connected back to back between the two machines, forming a closed loop as shown in Figure 8.

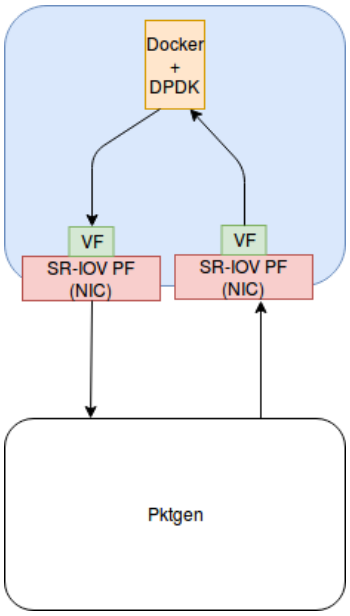

Figure 8: Test topology. DPDK based pktgen on one side and a Docker container with DPDK and SR-IOV on the other.

One of the machines is used as a packet generator. The other one utilizes virtualization of SR-IOV to push the packet directly to a container that runs a DPDK application. Hence, the packets received are not passed to the kernel network stack but rather handled by DPDK. The application simply forwards the packets from one port to the other. To run DPDK based application, SR-IOV Virtual Functions have to be enabled and CNI plug-ins need to be used to expose them to Docker container. Configurations of the host environment are described below.

For running DPDK based applications, Hugepages should be enabled in the

host machine. The Hugepage sizes that are supported in the most common Linux distributions are 2MB and 1GB. If 1GB Hugepages are to be used, it is recommended to have them configured at boot time of the host by setting kernel boot parameters. This it to ensure the availability of such large contiguous memory (1GB) in main memory. Allocating 1GB Hugepages is generally not possible at run time due to fragmentation of main memory. The kernel boot parameters can be set by editing `/etc/default/grub` and updating GRUB with `update-grub` command. For instance, edit `GRUB_CMDLINE_LINUX` to include, `GRUB_CMDLINE_LINUX="default_Hugepagesz=1G␣Hugepagesz=1G␣Hugepages=4"`. Then update GRUB, reboot machine and check boot parameters and consequently the Hugepage availability as follows,

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.4.0-131-generic default_Hugepagesz
   =1G Hugepagesz=1G Hugepages=4
$ cat /proc/meminfo | grep -i Hugepages_
Hugepages_Total:       4
Hugepages_Free:        4
Hugepages_Rsvd:        0
Hugepages_Surp:        0
Hugepagesize:    1048576 kB
```

Hugepages have to be mounted on the host. In some versions of Linux distributions, for instance, in Ubuntu 16.04 and higher, they are mounted automatically in `/dev/Hugepages`. It is also possible to isolate the CPU cores that will be used for the DPDK application using boot parameters so that the kernel scheduler does not use them. For instance, `GRUB_CMDLINE_LINUX="default_Hugepagesz=1G␣Hugepagesz=1G␣Hugepages=4␣isolcpus=2,3,4,5,6,7"`

DPDK supports different Linux kernel drivers such as UIO, VFIO, etc. [24]. VFIO is used in most of the cases in the thesis since, as stated in DPDK documentation, it is "more robust and secure". VFIO kernel module if supported, is shipped with the OS. It also needs IO virtualization such as Intel VT-d and that needs to be enabled in BIOS. I/O Memory Management Unit (IOMMU) has to be set as a kernel boot parameter,

```
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-4.4.0-131-generic default_Hugepagesz
   =1G Hugepagesz=1G Hugepages=4 intel_iommu=on
```

Setting up a SR-IOV interface for virtualization requires IOMMU and Intel VT-d support which are also needed when using VFIO driver as explained above. The number of VFs to be enabled can be set using `sys` virtual file system in `/sys/class/net/$IFNAME/device/sriov_numvfs` where `$IFNAME` is the name of the SR-IOV interface as displayed in `ip link` command. MAC address needs to be specifically set for each VF using `ip link set dev $IFNAME vf $i mac aa:bb:cc:dd:ee:01` where `$i` is the VF count. The Physical Function uses this MAC address to forward

a received packet to the correct VF and hence directly to the container or VM it is exposed to.

For using third party CNI plug-ins, when a Docker container is spawned, it looks for network configuration file in a default location, `/etc/cni/net.d/` and binary of the plug-in in `/opt/cni/bin/`. The plug-in used for this test case is SR-IOV CNI plug-in with support for DPDK [42].

The topology needs a DPDK based container on one side and a packet generator on the other. For building a container that can run a DPDK application, the idea is to start with a base image of some standard Linux distribution such as Ubuntu 16.04, install the required libraries and tools, including some debugging tools such as GNU debugger (GDB). Compilation of DPDK requires Linux header files and other standard build tools, make, etc. Then fetch the DPDK code, compile the libraries and the required application, `testpmd`. Some extra compile time flags are used to enable debugging with GDB.

```
$ git clone http://dpdk.org/git/dpdk dpdk
$ cat > build-dpdk.sh<<EOF
echo "Build␣DPDK"
make install T="\$RTE_TARGET" EXTRA_CFLAGS="-g␣-O0" -j8
if [ \$? != 0 ]; then
    echo -e "\tBuilding␣DPDK␣failed"
    exit 1
fi
cd app/test-pmd/
make -j8
EOF
$ chmod +x build-dpdk.sh
$ cat > Dockerfile <<EOF
FROM ubuntu:16.04
RUN apt-get update; apt-get install -y apt-utils build-
   essential less make kmod vim pciutils libnuma-dev python
   gdb linux-headers-$(uname -r);
COPY ./dpdk /root/dpdk
COPY ./build-dpdk.sh /root/dpdk/build-dpdk.sh
WORKDIR /root/dpdk/
ENV RTE_SDK "/root/dpdk"
ENV RTE_TARGET "x86_64-native-linuxapp-gcc"
RUN ./build-dpdk.sh
EOF
$ docker build -t dpdk-docker:latest .
```

Now, as is visible from Figure 8, one VF per SR-IOV interface is sufficient for this experiment. However, one VF from each SR-IOV interface is exposed to the container. The CNI plug-in configuration used is as follows,

```
$ cat /etc/cni/net.d/10-sriov-dpdk.conf
{
```

```
    "name": "net1",
    "type": "sriov",
    "if0": "enp1s0f1",
    "if0name": "if0",
    "dpdk": {
        "kernel_driver":"ixgbevf",
        "dpdk_driver":"igb_uio",
        "dpdk_tool":"\$RTE_SDK/usertools/dpdk-devbind.py"
    }
}
```

In the configuration above, `type` is always `sriov` for this plug-in, `if0` is the name of the SR-IOV interface or PF, `if0name` is the name inside the container and within dpdk parameters, `kernel_driver` is the current kernel driver the interface is using, `dpdk_driver` is the DPDK driver that needs to be used, such as VFIO and `dpdk_tool` is a tool that is shipped with DPDK source code that can be used to bind interfaces to DPDK. The above configuration could be replicated with different `if0` and `if0name` for the VF from second SR-IOV interface [42].

To run DPDK application in the container, the container has to be run in privileged mode so that it can bind the interfaces, get Hugepage information, etc. This can be done with the `--privileged` flag in Docker. Also, the Hugepage mount location has to be mounted in the container as a volume. Docker run command is as follows,

```
docker run -it -v /dev/Hugepages:/dev/Hugepages --privileged
   dpdk-docker:gdb /bin/bash
```

Inside the container, start testpmd application and configure it for forwarding packets received on one port to the other. If there are an even number of ports, this is also the default behavior of `testpmd`. This can be checked with `show config fwd` command. Start reception and transmission of packets with `start` command.

```
testpmd> show config fwd
io packet forwarding - ports=2 - cores=1 - streams=2 - NUMA
   support enabled, MP over anonymous pages disabled
Logical Core 1 (socket 0) forwards packets on 2 streams:
  RX P=0/Q=0 (socket 0) -> TX P=1/Q=0 (socket 0) peer
     =02:00:00:00:00:01
  RX P=1/Q=0 (socket 0) -> TX P=0/Q=0 (socket 0) peer
     =02:00:00:00:00:00
testpmd> start
```

On the packet generator side, the DPDK based packet generator `pktgen` requires similar configuration as above with the exception that there is no container. The VFs from SR-IOV interfaces are directly attached to the application. Latency has to be specifically enabled in both the interfaces and the packet size is set to 96 so that there is room in the packet buffer for time stamping. Here, we set the destination

MAC address as the MAC address of the receiving VF in DUT; if not, the packets get dropped at PF.

```
./app/x86_64-native-linuxapp-gcc/pktgen -l 0-3 -n 4 --proc-
   type auto --log-level 7 --socket-mem 2048 --file-prefix pg
    -- -T -P --crc-strip -m [1:2].0 -m [2:3].1 -f themes/
   black-yellow.theme
pktgen> enable 1 latency
pktgen> set 1 dst mac aa:bb:cc:dd:ee:00
pktgen> set 1 size 96
pktgen> start 1
```

Above, 1 refers to the port number used for generating packets. And with `start 1`, packets are generated at a rate of 11 million packets per second (Mpps). Latency was measured at the packet generator side to be 800 micro seconds. In comparison, the work conducted in "Supporting Fine-Grained Network Functions through Intel DPDK" [43] the latency was found to be 100 micro seconds for small number of VNFs. But this value quickly grew to about than 1890 micro seconds for higher number of VNFs. It should be noted that in [43], the latency was measured within the system and does not include wire latency. Also, different optimizations on DPDK were used to boost performance and the VNFs were implemented just as processes and not as containers. These are the probable reasons for a better latency value.

For a comparative study, instead of using DPDK based container running `testpmd`, the packets were processed using the Linux kernel stack. For this, a Linux bridge was created and the two SR-IOV interfaces were attached to it. The bridge forwards the traffic from one port to another. In this case, the latency was measured to be 8700 micro seconds for a packet rate of 10Mpps. It is clear that DPDK based packet processing provides much better performance, close to a 10 fold improvement.

4.2 Container on OVS-DPDK

The test setup in the previous section used was probably the simplest network possible. The machines were directly connected and the traffic blindly forwarded from one port to another. In a more realistic scenario, there could be multiple containers running on the host and packets forwarded between them. In a container orchestrated environment with Kubernetes, these containers or pods could very well reside in different hosts. This calls for exploring a more generic networking solution.

The default networking provided with Docker, using "docker0" bridge is a good solution that would work for containers in one host. Similarly, the overlay network driver in Docker and other third part network plug-ins such as using Flannel [44], Weave Net [47], Calico [48], etc. which also work with Kubernetes are also solutions that could work with little hassle. But none of these methods provide support for a fast data path using DPDK or support virtualization with SR-IOV VFs. However, Open vSwitch (OvS) on the other hand, supports DPDK and provides a way to easily configure the network using Open Flow. Though it might be an overhead in a simple topology, it could scale well when the network is more complicated with

multiple hosts and different types of interfaces such as for management and data since it also supports setting up tunneling interfaces to configure an overlay network which could be helpful with Kubernetes. However, whether the overlay network can support a DPDK based fast data path needs still to be explored.

This section uses OvS with DPDK and a container with DPDK application, `testpmd` on DUT. Both the SR-IOV interfaces are this time connected to the OvS bridge. The packet generator side is unchanged as the previous test scenario.

OvS release page [49] notes the compatible versions of DPDK with OvS and also the compatible version of OvS for different Linux kernel versions. In this experiment, the Linux kernel version used is `4.4.0-131-generic`. Correspondingly, the latest compatible version of OvS is `2.9.x` and DPDK version compatible this version of OvS is `17.11.3`.

To use Open vSwitch (OvS) with DPDK data path, DPDK needs to be built as in normal scenario but with the correct version as described above. The environment variables `RTE_SDK` and `RTE_TARGET` should be set. The former is the path to DPDK source code and the later is the build target environment, or `x86_64-native-linuxapp-gcc` in this case. Then OvS has to be configured with the flag, `--with-dpdk="$RTE_SDK/$RTE_TARGET"` and compiled from source.

DPDK parameters such as lcore, memory and other typical settings used with most other DPDK applications can be set in the OvS data base as shown below,

```
sudo ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk
    -init=true
sudo ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk
    -lcore-mask=0x0F
sudo ovs-vsctl --no-wait set Open_vSwitch . other_config:dpdk
    -socket-mem=256
sudo ovs-vsctl --no-wait set Open_vSwitch . external-ids:
    system-id="${HOSTNAME}id"
```

The system identifier, `system-id` above, could come in handy when setting up an overlay network with OvS. Most other steps in setting up OvS with DPDK are quite straightforward or well documented and not listed here.

The final part of the previous section had a Linux bridge with interfaces attached to it and packets forwarded as a normal bridge would do, flood. In this case, something similar in theory is done with an OvS bridge. But, the interfaces are bound to DPDK and then attached to an OvS bridge. Also couple of flows are added to the bridge to forward the packets received on one interface to the other and vice versa instead of flooding. The commands are shown in detail below.

```
$ ovs-vsctl add-br dpdk-br -- set bridge dpdk-br
    datapath_type=netdev
$ ovs-vsctl add-port dpdk-br p1 -- set Interface p1 type=dpdk
     options:dpdk-devargs=0000:01:00.0 ofport_request=1
$ ovs-vsctl add-port dpdk-br p2 -- set Interface p2 type=dpdk
     options:dpdk-devargs=0000:01:00.1 ofport_request=2
```

```
$ ovs-ofctl add-flow ext1 in_port=2,action=output:1
$ ovs-ofctl add-flow ext1 in_port=1,action=output:2
```

Packets were generated at an average of 10Mpps and the latency was observed to be 800 micro seconds. This is the same value of latency observed in the previous section 4.1 with SR-IOV VF exposed to a container running a DPDK application. This shows that OvS running as a DPDK application does not add any additional latency since OvS here is just a DPDK application. Also, counter intuitively, a container does not add any latency to packet processing. This is expected since containers utilize the host resources and do not have any overhead like a hypervisor in Virtual Machines.

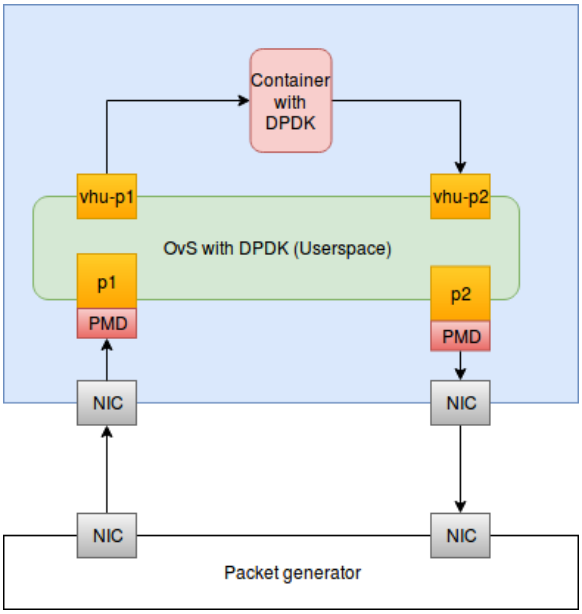

Figure 9: Test topology with OvS DPDK running on user space and forwarding packets to a container using vhostuser ports

Now, the next logical step is to add a container in between the two interfaces and forward the packets to a DPDK application, `testpmd`, running inside the container and forward the packets back to the packet generator as shown in Figure 9. To expose DPDK interfaces to a container via OvS, OvS uses `vhostuser` port on the host side. This creates a UNIX domain socket which needs to be exposed to the container. In DPDK application, a `virtiouser` port is created using this socket file. Now, flows are added to forward packets between the DPDK interface and `vhostuser` ports. The packets received at `vhostuser` will be picked by `virtiouser` and thus the DPDK application.

```
$ ovs-vsctl add-port dpdk-br vhu-p1 -- set Interface vhu-p0
   type=dpdkvhostuser
```

```
$ ovs-vsctl add-port dpdk-br vhu-p2 -- set Interface vhu-p1
   type=dpdkvhostuser
$ ovs-ofctl add-flow dpdk-br in_port=1,actions=output:3
$ ovs-ofctl add-flow dpdk-br in_port=4,actions=output:2
```

The port numbers in OvS could be verified with command `ovs-ofctl dump-ports-desc dpdk-br`. The UNIX domain sockets are created by default in location `/usr/local/var/run/openvswitch/`. So, this also has to be mounted while running DPDK Docker apart from the Hugepages mount, as shown below.

```
docker run --privileged -it -v /dev/Hugepages:/dev/Hugepages
   -v /usr/local/var/run/openvswitch:/var/run dpdk-docker:gdb
    /bin/bash
```

While running the DPDK application inside container, the PCI devices are disabled so that the application does not look for physical ports and virtual devices are created using the two `virtiouser` ports.

```
./testpmd -c 0xf -n 4 -m 1024 --no-pci --vdev=virtio_user0,
   path=/var/run/vhu-p1 --vdev=virtio_user1,path=/var/run/vhu
   -p2 --file-prefix=container -- -i
```

The latency measurement was done with the packet generator sending packets at a rate of 11 million packets per second. Latency was observed to be 5000 micro seconds. This is quite a high value when compared to the previous one with just the addition of a DPDK container. `vhostuser` and `virtio` are culpable for causing the overhead. Russell R., states that `virtio` implements ring buffers for packet copy between the Virtual Machines and the host[10]. In this case, this happens twice between the container application and host for transmission and reception of packets. Moreover, Bonafiglia et al, state that `vhostuser` waits for interrupts to be received from `virtio` on the container side and a tap interface on the OvS side before sending or receiving packets [50]. This is again another considerable overhead. The performance improvements by utilizing DPDK address the exact issues mentioned above, but the network bottleneck is just shifted with `virtio` and `vhostuser`. However, this is currently the most feasible solution especially with Virtual Machines since it emulates a PCI device and the hypervisor does not need extra configurations or drivers[10]. For a comparative understanding, the paper "Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks" [51] notes a latency of 39422 micro seconds with Virtual Machines which utilize the same `vhost` ports and `virtio` queues.

4.3 Fast packet processing in Kubernetes

The previous two sections dealt with the networking aspects when containers were hosted on just one host and packets were generated from another. But in a real

scenario, there would be a cluster of many hosts with some container orchestration such as Kubernetes managing them. When a request to deploy a new container or rather a pod, in Kubernetes is received, Kubernetes decides on which worker node to deploy depending on the requirements to run the pod such as available memory, CPU cores and any other constraints that may be provided in the pod configuration. Kubernetes has to find a node that fits all the bills. Fast data path could be also be another constraint to be considered while deploying pods.

In a Kubernetes cluster, a deployed pod needs one management interface to communicate with Kubernetes master [52]. As shown in Figure 1, one interface from each of the worker nodes and master is placed in the same subnet for this. When a pod is being deployed in Kubernetes and if it needs external DPDK interfaces for fast packet processing, Kubernetes has to decide on which worker node to deploy the pod depending on the availability of such interfaces in the worker nodes. There are three scenarios to be considered and are described below.

The first scenario is when the required number of DPDK interfaces for the pod is present in all of the Kubernetes worker nodes. In this case the pod could be deployed in any of the nodes and there is no extra networking constraint to be considered by the scheduler. The only question to be answered here is how to expose multiple interfaces to a pod since a Kubernetes pod in its default state supports only one interface.

The second scenario is when the required external DPDK interaces for the pod are present in just some of the worker nodes. In this case, the Kubernetes scheduler needs some mechanism to find out which nodes satisfy this criteria and deploy the pod in one of them.

The third scenario is the trickiest. If the required number of DPDK interfaces are say, three, and there are three worker nodes with one DPDK interface each. The decision on which node to deploy the pod is not important but rather routing the data between the worker nodes. Earlier it was noted that every Kubernetes pod has a default management interface. This could maybe leveraged to route the data packets as well.

The above three scenarios and their potential solutions are discussed in the next few sections. But the next immediate section is on how to build a Kubernetes cluster on bare metal.

4.4 Building a Kubernetes cluster

In this section, using bare metal machines, a cluster is formed with local connectivity, a DHCP server to assign dynamic private IP, and a DNS resolver. Then Kubernetes is run on this cluster. To build the cluster, MAAS or Metal as a Service, is used. MAAS is an open source software developed by Canonical [53]. It helps form a cluster from a set of bare metal machines with cloud like features and flexibility.

MAAS needs a central machine which is the controller. The controller is responsible adding further machines to the cluster, assigning IP address, booting them over a network interface using Preboot Execution Environment (PXE) booting, etc. The other nodes are generally accessible through the controller. The topology is shown in

Figure 10. The easiest way to setup the MAAS controller is using a Ubuntu server image and in the install options, choosing MAAS region controller. Another option is to install standard Ubuntu distribution, use the package manager, aptitude and install MAAS. The next step is to configure the controller. Most of the configurations are done using the HTTP server that runs by default on port 5240 on the controller. A MAAS cluster can have two types of controllers, region and rack controller. Since the setup is quite small and the machines are physically attached to the same sub network, just one controller is used which acts both as region and rack controller.

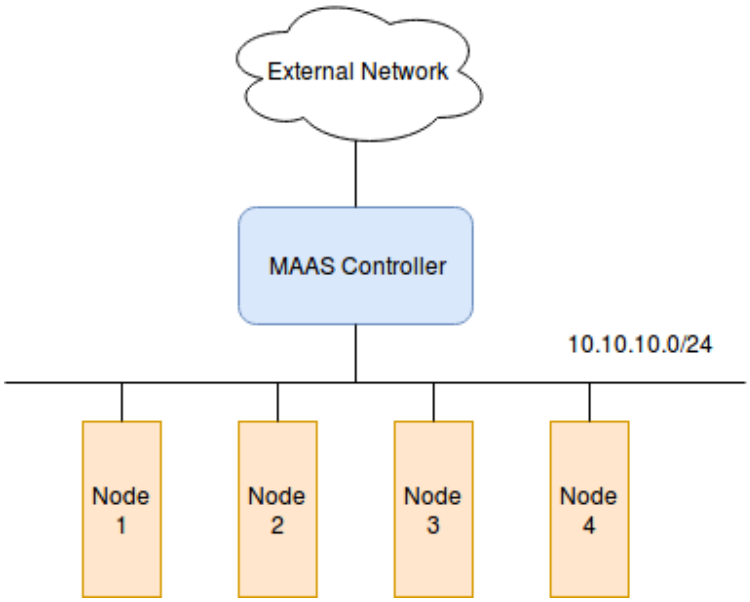

Figure 10: Topology of cluster built with MAAS

Initially, the controller SSH public key will need to be provided to MAAS. This is later used to allow access to the controller and the other nodes in the cluster. The version of Ubuntu to be used for deploying the nodes was set to 16.04. For upstream name resolution, the upstream Domain Name System (DNS) configuration can be set. Also, since the idea is to run DPDK based applications on Kubernetes and since some nodes will support SR-IOV interfaces, the required Kernel boot parameters can be set in MAAS controller API.

```
intel_iommu=on iommu=pt default_Hugepagesz=1G Hugepagesz=1G
    Hugepages=4
```

To allow access to the external network for other nodes in the cluster, the controller will have be configured to use NAT to forward the traffic. This is used only for remote login to the worker nodes and for installing other required packages and not for fast data path. For convenience, let the interface connected to the external network in the controller be `eth0` and the interface connected to the subnet `10.10.10.0/24` be `eth1`. The `iptable` configurations for NAT'in are as shown below.

```
iptables --table nat --append POSTROUTING --out-interface
   eth0 -j MASQUERADE
iptables --append FORWARD --in-interface eth1 -j ACCEPT
echo 1 > /proc/sys/net/ipv4/ip_forward
```

Other nodes to be added to the cluster should be placed in the sub-net. A switch could be connected to the MAAS controller interface `eth1`. Other nodes should be connected to the switch using the interface with PXE boot enabled. On booting, the node gets a temporary IP address from the DHCP server running on the controller. The controller then lists the node in the MAAS dashboard. The power type in the node has to be configured. If the node supports IPMI, it will be detected automatically. Otherwise, set the power type to "Manual". The disadvantage is that, the machines will have to be powered on/off manually. Now, in the MAAS dashboard, it is possible to commission the node. The machine will boot again, or will have to be done so manually, after which the controller sends the boot images and initialization scripts. The machine finally goes to ready state.

Once all the nodes are added and in "Ready" state, they can be acquired while deploying some application in MAAS. The application of interest here is Kubernetes and the easiest way to deploy Kubernetes on MAAS is using Juju. A Kubernetes deployment typically has many components that need to be installed. For instance, in the master node, "etcd", "easyrsa" for generating certificates for TLS, kubectl application, some network plug-in and of course the core Kubernetes applications such as scheduler, config manager, etc. On the worker nodes, Docker, kublet, the proxy daemon, etc. Juju does an automatic deployment of all the necessary components. But for this, Juju needs to be deployed first and it needs a dedicated controller. One way to do so without consuming an additional machine is to use a virtual machine in MAAS controller and add it to the cluster. It could then be used as Juju controller. The other way is to use any other node in MAAS cluster as Juju controller. The choice depends on how machines are available and how many are needed for Kuberentes. In this experiment, the virtual machine method was used initially. Juju can deploy a minimal Kubernetes cluster with two machines using the command, `juju deploy cs:bundle/kubernetes-core-251`. Verify the status of machines go to "active" with the following command and sample output

```
$ juju status
Model    Controller  Cloud/Region  Version  SLA
default  test-cloud  test-cloud    2.3.8    unsupported

App                Version  Status  Scale  Charm              Store       Rev  OS      Notes
easyrsa            3.0.1    active      1  easyrsa            jujucharms   27  ubuntu
etcd               2.3.8    active      1  etcd               jujucharms   63  ubuntu
flannel            0.9.1    active      2  flannel            jujucharms   40  ubuntu
kubernetes-master  1.9.8    active      1  kubernetes-master  jujucharms   78  ubuntu  exposed
kubernetes-worker  1.9.8    active      1  kubernetes-worker  jujucharms   81  ubuntu  exposed

Unit                  Workload  Agent  Machine  Public address  Ports           Message
easyrsa/0*            active    idle   0/lxd/0  10.10.10.8                      Certificate Authority connected
.
etcd/0*               active    idle   0        10.10.10.6      2379/tcp        Healthy with 1 known peer
kubernetes-master/0*  active    idle   0        10.10.10.6      6443/tcp        Kubernetes master running.
  flannel/0*          active    idle            10.10.10.6                      Flannel subnet 10.1.18.1/24
kubernetes-worker/0*  active    idle   1        10.10.10.7      80/tcp,443/tcp  Kubernetes worker running.
  flannel/1           active    idle            10.10.10.7                      Flannel subnet 10.1.34.1/24

Machine  State    DNS         Inst id              Series  AZ       Message
```

```
0        started  10.10.10.6  yda8rf               xenial  default  Deployed
0/lxd/0  started  10.10.10.8  juju-90b946-0-lxd-0  xenial  default  Container started
1        started  10.10.10.7  6hn4pg               xenial  default  Deployed
```

Following is a hello world example to test the Kubernetes deployment. To deploy a simple pod in Kubernetes, create pod manifest file and use `kubectl` commands to create the pod.

```
$ cat > helloworld.yaml <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: test-pod
spec:
  containers:
  - name: test-container
    image: ubuntu:latest
    command: ["/bin/bash"]
    args: ["-c", "echo␣SUCCESS;␣while␣true;␣do␣sleep␣1;␣done"
       ]
EOF
$ kubectl create -f helloworld.yaml
$ kubectl get pods
NAME             READY     STATUS         RESTARTS    AGE
test-pod         1/1       Running        0           5s
$ kubectl logs test-pod
SUCCESS
```

Another way to deploy Kubernetes is using the tool `kubeadm`. This is a Kubernetes project that is still in beta stage but quite good for development purposes. This deployment of Kubernetes is quite minimalistic when compared to deployment with Juju. This gives a bit more control. With `kubeadm`, the network plug-in has to be installed separately which is ideal for most of the work in the thesis since it can be easily tweaked and modified.

4.5 Scenario 1, Multus

This section covers in detail the first scenario described above where all the worker nodes in the Kubernetes cluster have DPDK supported interfaces. The scheduling decision is hence unaffected by the worker nodes' hardware features and the normal scheduling decision could be made based on the CPU, memory and other resource constraints.

To use the DPDK interfaces in a pod, a Container Network Interface (CNI) plug-in called "Multus" can be used. It is an open source software developed by Intel [54] and written in Golang. Multus acts as an interface between Kubernetes and other CNI plug-ins. It reads the required networking requirement from a configuration file which

specifies which other CNI plug-ins to use along with their individual configuration. It then looks for these CNI plug-in executable in a specific location and runs them with the given configuration. Consequently exposing the CNI plug-in based interfaces to the pod. Multus also maintains one interface for Kubernetes related communication. This is called the master plug-in. In essence, if CNI plug-in, say P1 needs to be called with configuration C1 and another CNI plug-in P2 with configuration C2, the user gives these configurations to Multus and Multus calls these individual CNI plug-ins one after the other, exposing it to the pod.

The CNI plug-in for SR-IOV with DPDK which was used in section 4.1 and Flannel for management interface are used with Multus in this scenario. Here Flannel is the master plug-in used in Multus nomenclature and SR-IOV DPDK plug-in is used for fast packet processing and latency measurements.

There are a couple of different ways of using Multus. Kubernetes allows users to define custom resource types, called Custom Resource Definition (CRD). The first approach involves defining a new network resource as a CRD and specifying the plug-in configurations while defining the resource objects. The other option is to have a CNI configuration file in each of the worker node with the required configuration. The second method has a disadvantage that all the pods running in a worker will use the same configuration file and it is not possible to customize the network per pod. Since a generic solution is really not needed for the targeted latency measurements, the second approach is used. Multus configuration file is as follows.

```
$ cat /etc/cni/net.d/10-multus.conf
{
    "name": "multus-demo-network",
    "type": "multus",
    "delegates": [
       {
            "type": "flannel",
            "masterplug-in": true,
            "delegate": {
                "isDefaultGateway": true
            }
        },
        {
            "type": "sriov",
            "name": "net1",
            "if0": "enp1s0f1",
            "if0name": "if0",
            "dpdk": {
                "kernel_driver":"ixgbevf",
                "dpdk_driver":"igb_uio",
                "dpdk_tool":"\$RTE_SDK/usertools/dpdk-devbind
                   .py"
            }
        },
        {
```

```
            "type": "sriov",
            "name": "net2",
            "if0": "enp1s0f2",
            "if0name": "if1",
            "dpdk": {
                "kernel_driver":"ixgbevf",
                "dpdk_driver":"igb_uio",
                "dpdk_tool":"\$RTE_SDK/usertools/dpdk-devbind
                    .py"
            }
        }
    ]
}
```

The experiment in this section is similar to the previous sections with a packet generator on one side sending and receiving packets and calculating the latency. But this time on the DUT side is a Kubernetes cluster with SR-IOV interfaces from the packet generator directly connected to one of the worker nodes. The SR-IOV VFs are exposed to the Kubernetes pod using Multus and other CNI plug-ins as shown in Figure 11. Here the idea is to measure the latency when all the Kubernetes worker nodes have DPDK interfaces. The experiment is quite similar to the one in Section 4.1 but instead of a Docker container, a Kubernetes pod is used and Multus is used for multiple interfaces.

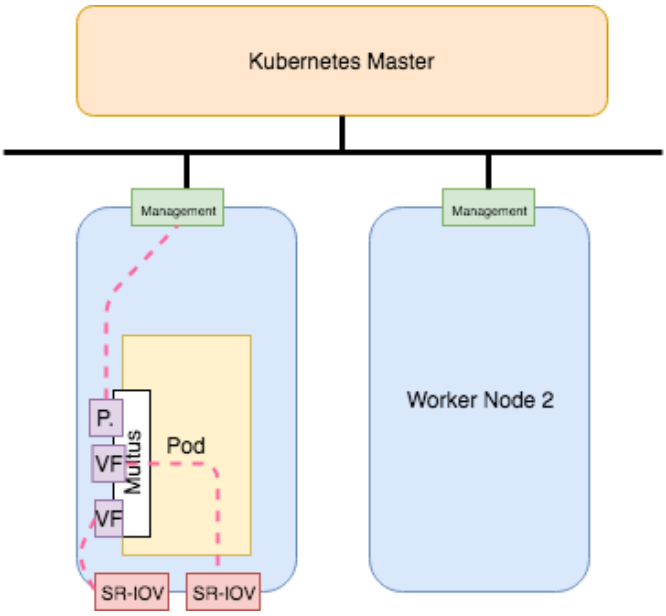

Figure 11: Test topology. DPDK based pktgen on one side and a Kubernetes cluster with multiple SR-IOV VFs exposed to the pod using Multus.

The latency was found to be 800 micro seconds, similar to the values in Section 4.1. This is quite expected since the CNI plug-ins just expose the interfaces directly

to the pod. There is not extra packet copy or any processing induced by them. It should be noted that the SR-IOV interfaces here are not managed by Kubernetes and exposed externally and as a result gives "native" DPDK performance. In the later section 4.7, the overhead caused due an overlay network with packets routed between nodes is understood.

4.6 Scenario 2, Node feature discovery

In this scenario, only some of the Kubernetes worker nodes may have the required number of DPDK interfaces. To deploy the pod in one of such nodes, the Kubernetes scheduler should have some mechanism to find out the interfaces supported in each of the nodes.

Kubernetes supports "daemon set" as a type of work load which ensures that the pod runs on all the nodes in Kubernetes. This can be used to figure out the hardware features of the node and report them to the API server. The scheduler can then use this information to decide which node can run the pod.

There is a Kubernetes incubator project called "Node Feature Discovery" [37] that does exactly this, discovering different hardware features supported by the worker nodes. As of writing, it supports finding out if a node has SR-IOV interfaces. This could be used along with a modified scheduler to deploy the node. A similar task is done in the chapter 5 and not repeated here.

4.7 Scenario 3, Kubernetes with OVN

In the final scenario, the required number of DPDK supported interfaces are not found in one host. It could be scattered over the cluster. For instance, consider Figure 1 in Page 10. If the application running on the pod needs four interfaces, two for incoming packets, process them and send them over to another network using the other two. The pod could very well be placed in any of the worker nodes but none of them satisfy the required number of interfaces individually. So, the focus here is on finding a way to route the data packets within the cluster.

One way to achieve this is to add dedicated interfaces between the nodes apart from the management interfaces that support DPDK and provide fast data path. But this imposes extra hardware requirement on the cluster, increases cost and is generally not a good solution. The other solution is to find a way to use the management interfaces for routing the data packets.

In a typical Kubernetes deployment that uses some networking plug-in such as Flannel, the requirement that each pod in Kubernetes is reachable via an IP address is taken care of by the plug-in. Flannel takes care of networking between the nodes in the cluster. A subnet of the entire allocable IP address is provided to each host [44]. A solution applicable to the scenario here would on top of this, place the external interfaces and pods in a different private IP subnet and use some tunneling protocol to route the packets between them. This calls for complicated routing within the cluster and a more configurable network is handy.

Open Virtual Network (OVN) can be used for building such a network. OVN can be used to create a virtual network and provide Layer 2 and 3 connectivity. Pods could be placed in this virtual network and provide connectivity similar to what Flannel does. OVN is like a controller for a virtual network built on OvS comprising abstractions such as logical switches and routers. It can be used to build a full scale virtual network with OvS lying underneath it. The OVN architecture [45] has at the top a Cloud Management System (CMS) through which it receives the network configuration. CMS populates Northbound database with the configurations. Below that is a Northbound interface that connects this database to the Southbound database. It converts the logical network configuration to logical flows. Apart from these logical flows, Southbound database also contains some physical network information. This is in turn connected to OVN controller running on each node. These controllers are connected to `ovs-vsctl` to configure the OvS bridges underneath.

At the Northbound interface, OVN accepts configurations in terms of logical switches and routers. For instance, to provide connectivity between pods in Kubernetes, a configuration comprising logical switches on all hosts and connected together using a logical distributed router would suffice[1]. OVN uses gateway routers to connect to external networks. For two routers to be connected to one another, a logical switch is needed between them, at the time of writing.

OVN has plug-ins to function with Open Stack and Kubernetes [55]. The plug-in for Kubernetes sets up the Kubernetes networking requirements similar to Flannel and provides further possibility to build the network. Most importantly, since underneath it uses OvS, DPDK could be used for data path.

For Kubernetes, kubeadm based deployment is used instead of Juju as described in section 4.4. This is because kubeadm gives more flexibility in configuring the cluster components and network plug-ins are not installed by default. To install kubeadm, use updated package manager [56] as shown below.

```
$ curl -s https://packages.cloud.google.com/apt/doc/apt-key.
   gpg | sudo apt-key add -
$ sudo su -
$ cat <<EOF > /etc/apt/sources.list.d/kubernetes.list
deb http://apt.kubernetes.io/ kubernetes-xenial main
EOF
$ logout
$ sudo apt-get update
$ sudo apt-get install -y docker.io kubeadm
```

To start kubeadm, swap must be disabled and use the `init` command with the required configuration.

```
$ sudo swapoff --all
$ cat <<EOF > kubeadm.yaml
kind: MasterConfiguration
apiVersion: kubeadm.k8s.io/v1alpha1
controllerManagerExtraArgs:
```

```
  horizontal-pod-autoscaler-use-rest-clients: "true"
  horizontal-pod-autoscaler-sync-period: "10s"
  node-monitor-grace-period: "10s"
apiServerExtraArgs:
  runtime-config: "api/all=true"
kubernetesVersion: "stable-1.11"
EOF
$ sudo kubeadm init --config kubeadm.yaml
```

After installation, the logs print out the kubeadm join command that can be used in other nodes to join this cluster. To get OVN to work with Kubernetes, some default Role Based Access Control (RBAC) that Kubernetes uses needs to be modified. The following permissions for role `system:node` should be present so that the worker nodes can, for instance, watch "endpoints" and "network policies".

```
$ kubectl edit clusterrole system:node
- apiGroups:
  - ""
  resources:
  - endpoints
  - namespaces
  verbs:
  - get
  - list
  - watch
- apiGroups:
  - networking.k8s.io
  resources:
  - networkpolicies
  verbs:
  - get
  - list
  - watch
```

Edit role binding for `system:node` and `subjects` so that the above permissions take effect for the worker nodes.

```
$ kubectl edit clusterrolebinding system:node
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: Group
  name: system:nodes
```

To use OVN CNI plug-in for Kubernetes, OvS needs to be installed first. For this, the process described in Section 4.2 is followed, with DPDK support. To set up OVN in the master, start both Northbound daemon and OVN controller. Then install and start the Kubernetes plug-in.

```
$ export CLUSTER_IP_SUBNET=192.168.0.0/16
$ export NODE_NAME=$HOSTNAME
$ export SERVICE_IP_SUBNET=10.96.0.0/12
$ sudo mkdir -p /var/log/openvswitch
$ sudo touch /var/log/openvswitch/ovnkube.log
$ export CENTRAL_IP=10.10.10.24
$ export TOKEN=u533e5.9ml2dhgng8aaesra
$ nohup sudo ovnkube -k8s-kubeconfig $HOME/.kube/config -net-
   controller \
 -loglevel=4 \
 -k8s-apiserver="http://$CENTRAL_IP:8080" \
 -logfile="/var/log/openvswitch/ovnkube.log" \
 -init-master=$NODE_NAME -init-node=$NODE_NAME \
 -cluster-subnet="$CLUSTER_IP_SUBNET" \
 -service-cluster-ip-range=$SERVICE_IP_SUBNET \
 -nodeport \
 -k8s-token="$TOKEN" \
 -nb-address="tcp://$CENTRAL_IP:6641" \
 -sb-address="tcp://$CENTRAL_IP:6642" 2>&1 &
```

In the above commands, some environment variables are set and passed to OVN Kubernetes plug-in. `CLUSTER_IP` is private IP address space for the cluster. `NODE_NAME` is set to the hostname of the machine. `SERVICE_IP_SUBNET` is taken from Kubernetes API server command **`ps -aef | grep kube-apiserver`**. `CENTRAL_IP` is the IP of the master node which is reachable to the worker nodes as well. And the value of `TOKEN` can be taken from kubeadm join command.

Once the OVN CNI plug-in is running, the status of node in the command `kubectl get nodes` change to "Ready". To start the plug-in in worker nodes,

```
$ nohup sudo ovnkube -k8s-kubeconfig $HOME/.kube/config -
   loglevel=4 \
    -logfile="/var/log/openvswitch/ovnkube.log" \
    -k8s-apiserver="http://$CENTRAL_IP:8080" \
    -init-node="$NODE_NAME"  -nodeport \
    -nb-address="tcp://$CENTRAL_IP:6641" -sb-address="tcp://
       $CENTRAL_IP:6642" \
    -k8s-token="$TOKEN" -init-gateways -service-cluster-ip-
       range=$SERVICE_IP_SUBNET \
    -cluster-subnet=$CLUSTER_IP_SUBNET 2>&1 &
```

OVN Kubernetes plug-in creates a distributed router in the master node. This router is the main component responsible for routing within the cluster. On every worker node, there is a logical switch which is allocated a subnet of `CLUSTER_IP_SUBNET` This logical switch is connected to the distributed router to the top and an OvS bridge below. The OvS bridge is always named `br-int`. This in turn is connected to the pods similar to "docker0" bridge. There are also gateway routers created in the worker

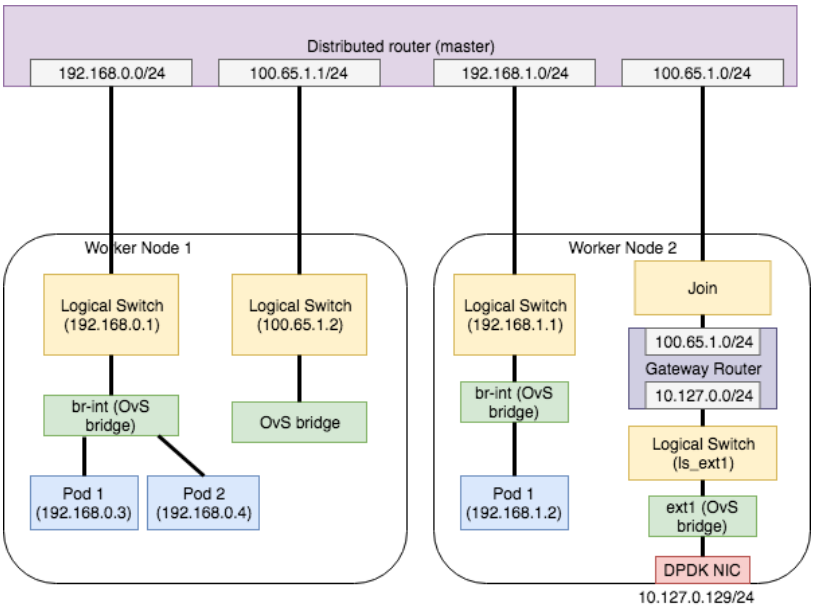

Figure 12: OVN based network in Kubernetes with external DPDK interface

nodes for connecting the OVN controller to the Northbound interface. The logical network topology can be viewed from master using command `ovn-nbctl show`.

To add external DPDK interfaces to OVN, gateway routers are used. The gateway router is connected via a logical switch and an OvS bridge to the physical interface. On top it is connected to the distributed router. For routing of the packets received at the physical interface, NAT is used to map the physical interface address to a private IP subnet. Then, ports are added to the distributed router in master and the logical switches in all worker nodes in the same sub-net as shown in Figure 12. The configurations are shown below.

```
// Add GW router on worker node 2 (l0) and add ports
$ sudo ovn-nbctl lr-add gw_ext1
$ sudo ovn-nbctl lrp-add gw_ext1 gw_ext1_down aa:bb:cc:dd:ee
   :01 10.127.0.129/24
$ sudo ovn-nbctl lrp-add gw_ext1 gw_ext1_up aa:bb:cc:dd:ee:02
    100.65.1.2/24
$ sudo ovn-nbctl lrp-set-gateway-chassis gw_ext1_up l0id
$ sudo ovn-nbctl lrp-set-gateway-chassis gw_ext1_down l0id

// Add port in distributed router in master (l5)
$ sudo ovn-nbctl lrp-add l5 l5-ext1 aa:bb:cc:dd:ee:05
   100.65.1.1/24

// Add join logical switch
$ sudo ovn-nbctl ls-add join_ext1
$ sudo ovn-nbctl lsp-add join_ext1 join_ext1_up
```

```
$ sudo ovn-nbctl lsp-add join_ext1 join_ext1_down
$ sudo ovn-nbctl lsp-set-type join_ext1_up router
$ sudo ovn-nbctl lsp-set-type join_ext1_down router
$ sudo ovn-nbctl lsp-set-addresses join_ext1_up aa:bb:cc:dd:
   ee:03
$ sudo ovn-nbctl lsp-set-addresses join_ext1_down aa:bb:cc:dd
   :ee:04
$ sudo ovn-nbctl lsp-set-options join_ext1_down router-port=
   gw_ext1_up
$ sudo ovn-nbctl lsp-set-options join_ext1_up router-port=
   l5_ext1

// Logical switch towards the external network
$ sudo ovn-nbctl ls-add ls_ext1
$ sudo ovn-nbctl lsp-add ls_ext1 ls_ext1_up
$ sudo ovn-nbctl lsp-add ls_ext1 ls_ext1_down
$ sudo ovn-nbctl lsp-set-type ls_ext1_up router
$ sudo ovn-nbctl lsp-set-addresses ls_ext1_up aa:bb:cc:dd:ee
   :06
$ sudo ovn-nbctl lsp-set-options ls_ext1_up router-port=
   gw_ext1_down

// Create bridge for localnet in l0
$ sudo ovs-vsctl add-br ext1 -- set bridge ext1 datapath_type
   =netdev
$ sudo ovs-vsctl set Open_vSwitch . external-ids:ovn-bridge-
   mappings=sriov1:ext1

// Configure  the localnet port
$ sudo ovn-nbctl lsp-set-addresses ls_ext1_down unknown
$ sudo ovn-nbctl lsp-set-type ls_ext1_down localnet
$ sudo ovn-nbctl lsp-set-options ls_ext1_down network_name=
   sriov1
```

When OVN Kubernetes CNI plug-in is used, it exposes an interface to a pod which is the management interface used by Kubernetes. With the above configurations, it is possible to route packets from external DPDK interface to another worker node which receives it at an OvS bridge. To expose a port from this bridge to the pod along with the OVN management interface, Multus needs to be used. But currently this is not possible since the CNI configuration file for OVN plug-in does not distinguish the underlying OvS bridge. Hence, the measurements were carried out in a similar manner to the first experiment in Section 4.2 where the OvS bridge just forwards the packets blindly to the other port without a pod in between. For a packet rate of 11 Mpps, the latency was found to be 10000 micro seconds.

The latency in this setup is quite high and not a good option for NFV. The extra hop between the worker nodes obviously adds to the latency. But this does not paint the whole picture. The gateway router uses Network Address Translation

to map the external IP address to the internal private IP address used for routing between the nodes. Moreover, only the OvS bridge runs on DPDK, the tunneling in the host interfaces connecting multiple worker nodes and master do not. It is possible to overcome this by separating Kubernetes management interfaces between the hosts and using extra interfaces between the worker nodes as shown in Figure 13. But this calls for extra networking hardware requirements and counter productive to the flexibility offered by Kubernetes. A similar study was done in [51], the overlay network was built using OvS and further optimized using real time kernel. The latency varied from 157 to 647 micro seconds. The network latency does not include wire latency but just time spent in the system processing packet. The extra abstractions in OVN on top of OvS is also another overhead and in hindsight, OVN is probably more suited as a Software Defined Networking (SDN) controller than for building a performance critical network.

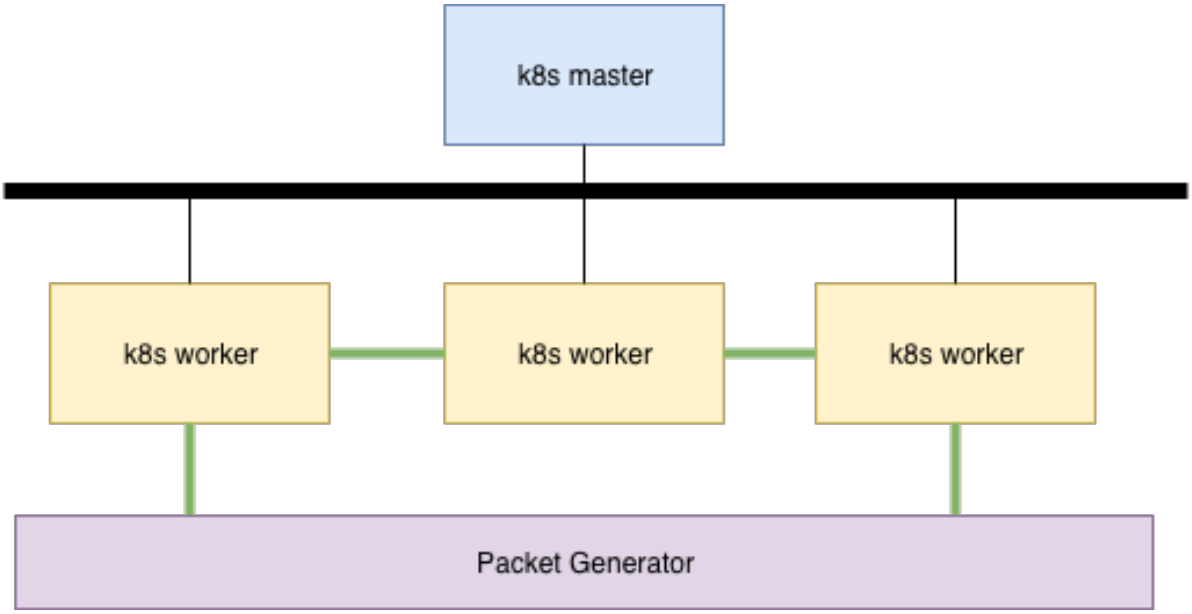

Figure 13: An alternative solution to using overlay network in Kubernetes. Dedicated DPDK interfaces (in green) between the worker nodes for fast data path.

To conclude, using OVN with Kubernetes does make the network much more configurable which is not possible while using other off-the-shelf networking solutions for Kubernetes such as Flannel [44], Calico [48]. But the overhead caused due to extra layer of abstractions and forwarding data packets between different hosts is probably not suited for a latency sensitive application. Also the solution used above requires lot of manual configurations right now. Since the Kubernetes OVN plug-in is open source, it is possible to automate them and also add support to be used with Multus.

4.8 Results and Observation

This chapter so far has explored various fast data path solution for a container in stand alone and cluster environments. The latency values are summarized in Figure 14. Based on the results, it is clear that some of the solutions provide much better latency results. The latency introduced in some of the solutions were software induced and in some others, due to the extra hops between the nodes. When DPDK is used in the "native" mode, which includes a container running a DPDK application, OvS running on DPDK and also with Multus and SR-IOV CNI plug-ins exposing the interfaces directly to a DPDK based pod, the latency was found to be 800 micro seconds. This is quite a satisfactory result for NFV applications [58]. Importantly, it was observed that it is possible to attain the same performance in orchestrated environments such as with Kubernetes, by exposing the interfaces externally using CNI plug-ins.

	Topology	Latency (micro seconds)
Stand alone	Linux Bridge	8700
	DPDK - Container	800
	OvS + DPDK	800
	OvS - DPDK + DPDK - Container	5000
Kuber netes	Multus + DPDK - Pod	800
	OVN	10000

Figure 14: Results of latency measurements for stand alone and orchestrated environments

It was also observed that with the addition of a container running DPDK application along with OvS on DPDK induced some overhead in packet processing. The reason for this, as noted in 4.2 could be due to `vhost` and `virtio` queues. The latency here is a six fold increase compared to "native" DPDK performance but still a much better result than using the default Linux kernel network stack where the latency was 8700 micro seconds. OVN was used to configure an overlay network within the cluster to route data packets between different hosts. The latency in this case was quite high and found to be not suitable for low latency NFV applications. This implies that scenario 3 described in 4.7 is not a practical solution. As a result, if a pod running NFV application needs multiple external interfaces, at least one

of the Kubernetes worker nodes should support that many interfaces. Section 4.6 describes a way to find out the hardware features available at each worker node. This information has to be provided to the Kubernetes scheduler so that a NFV pod is deployed on the correct worker. Also, current Kubernetes scheduler has to be extended to process this information. The next section describes a Kubernetes scheduler that achieves this.

5 Latency based scheduling

This section is an attempt to utilize the results obtained in the previous section while deploying pods in a cluster so that their network latency requirements can be met. Depending on the latency requirement provided at the time of creating a pod, the right host is chosen based on the network hardware features available. As noted in 2, the current schedulers for container orchestration systems such as Kubernetes and Apache Mesos do not consider network latency while scheduling the pods. This is not quite suitable for NFV where some pods could be running CPU or memory intensive applications while other have stringent network latency requirements. While deploying the pods, it is necessary to consider network latency as well. Consider Figure 1. For instance, if all the nodes have same CPU and memory resources available, but only some or one of them, such as Node 1, have DPDK or SR-IOV interfaces, the default scheduler could use this node for pods that are not network latency sensitive applications.

5.1 A Kubernetes Scheduler

The default scheduler in Kubernetes considers only CPU and memory resources in the worker nodes while scheduling pods [59]. This implies a new method to provide the latency requirement for a pod has to be found and also a scheduler capable of implementing this.

Annotations in Kubernetes could be used to specify the latency requirements for a pod during deployment. As per the Kubernetes documentation [39], annotations are used for specifying meta-data and can be retrieved using the client libraries. For this purpose, the latency requirement is given as a `key` and `value` pair.

```
annotations:
  latency : "5000"
```

In Kubernetes scheduler, there are multiple steps involved while deciding to which node a pod should be bound. In the first step, the scheduler filters out the incompatible nodes. For instance, nodes that do not have enough CPU or memory. Then, from the compatible nodes, the scheduler assigns priority to them while trying to distribute the work load among all the worker nodes. In the last step, the scheduler binds the pod to the node with the highest priority and informs the API server [60].

There are different ways to affect the scheduling decision in Kubernetes. One way is to modify the Kubernetes source code and implement the required features. Another is to implement a Kubernetes scheduler extender which exposes an IP address and port that is accessible to the default scheduler. The extender is called once the default scheduler is done with the first two steps mentioned above, filtering and prioritizing. This is like an extra step in the scheduling process. The final approach is to write a new scheduler that replaces or works along with the existing default scheduler. This new scheduler could be run as a pod in the cluster itself. The pod that wishes to use this scheduler could specify so in the manifest with `schedulerName:`

The source code of the scheduler implemented for the thesis is listed in Appendix A. The scheduler communicates with the Kubernetes API server using the `kubeconfig` file or it could be run as a pod in the cluster. It initially gets a list of nodes in the cluster and also all the pods that are configured to use scheduler with name `latency-scheduler`. It then runs the scheduling algorithm, finds the node that satisfies the latency requirement and sends a "Bind" message to the API server to bind the pod to that node. From a pod perspective, to use this scheduler, the scheduler name and latency requirements have to be specified in the manifest.

The second important factor to be addressed for the scheduler to work is to discover the hardware features of the worker nodes; specifically the support for DPDK and SR-IOV interfaces. For this, a daemon set is used [40]. A Daemon set runs on all the nodes in Kubernetes. This could be utilized to fetch the interface hardware information and report to the API server. The code to implement this daemon set is listed in Appendix B along with the daemon set manifest file and the Dockerfile to build a Golang container with the application. The code currently adds a new `map` entry of `dpdk:<number-of-dpdk-interfaces>` to `node.Labels` [61] where `node` is an object of the `Node` structure in Kubernetes Core v1. This information is in turn used by the scheduler.

```
if required_latency less than least latency possible with
   Multus:
    return error "Latency␣requirement␣cannot␣be␣met"
else if required_latency greater than least latency possible
   with Multus and less than latency with hops within the
   same host using OvS (for a NFV service chain):
    bind_pod(node with required multiple interfaces)
else if required_latency greater than latency with OvS and
   less than a default value:
    bind_pod( FindNode(nodes, LATENCY_WITH_OVS)
else if required_latency greater than default value:
    bind_pod(to any worker node)
```

In the scheduling algorithm, if the latency requirement is between 800 and 5000 micro seconds, the pod is bound to a host that has DPDK interfaces directly attached to it. The value of `node.Labels` is then reduced and updated at the API server so that in the next run, the available number of interfaces is correct. If the latency requirement is less that 800 micro seconds, the pod is not scheduled and the scheduler throws an error. Similarly, if the pod latency is between 5000 and 10000 microseconds, it is scheduled to a host with OvS. For higher values, it can be scheduled to any of the hosts. This host is chosen in a random manner from all the hosts that do not have DPDK interfaces, as those can be saved for low latency applications that may come up later. Snippet of scheduling algorithm is shown in above and the complete source code is provided in A.

6 Conclusion and Future Work

To conclude, the thesis explores various fast data path solutions for containers in stand alone and orchestrated environments. Experiments were carried out for various scenarios primarily utilizing technologies such as DPDK, SR-IOV, OvS and OVN. The experiments focused on measuring the network latency in these scenarios. It was found that native DPDK performance could be achieved in Kubernetes by exposing interface directly to pods using Multus. Some of the other solutions using OvS and OVN did not give the same performance but provided a more flexible networking solution as a whole. These results were further utilized to deploy pods in Kubernetes in an intelligent way by taking into account their network latency requirements. There is no direct comparison possible of this approach since the existing scheduler does not take network latency into account at all and deploys the pods based on other factors such as CPU and memory pressure on worker nodes.

There are several topics that could be explored further as enhancements to the work done in this thesis. Optimizing DPDK applications for Kubernetes is one of them. The number of CPU cores used, the Rx and Tx buffers per core, Hugepage allocation, etc., could be optimized especially when there are multiple DPDK containers in one host. Also Kubernetes is not NUMA aware and NUMA locality could increase performance.

Docker does not support Hugepage isolation. When there are multiple DPDK based containers running on the same host, Hugepage isolation is important as some applications may end up consuming all the free Hugepages. This could be implemented using Python SDK provided by Docker to accept an extra argument in docker for the amount of Hugepages to be allocated and some Python cgroup library to isolate them.

Another topic is, as explained in "Minimizing latency of Real-Time Container Cloud for SoftwareRadio Access Network" [51], using a real time kernel. This could be fruitful in improving packet processing latency. And finally, getting OVN Kubernetes plug-in working with Multus and also automatically adding the external interface to a private sub-net which was done manually in this thesis.

References

[1] Bl, H. et al. (2016). *DPDK-based Improvement of Packet Forwarding*. ITM Web of Conferences.

[2] Molnar L. et al. (2016). *Dataplane Specialization for High-performance OpenFlow Software Switching*. SIGCOMM, Brazil.

[3] Yu, T. et al. (2016). *FreeFlow: High Performance Container Networking*. Association for Computing Machinery Workshop on Hot Topics in Networks, USA.

[4] Hong, D. et al. (2017). *Considerations on Deploying High-Performance Container-based NFV*. Cloud-Assisted Networking Workshop, Incheon, Republic of Korea.

[5] Gorman, M., (2010). *Hugepages*. Available at https://lwn.net/Articles/374424/. [Accessed: 10.05.2018].

[6] Pongracz, G. et al. (2013). *Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK*. European Workshop on Software Defined Networks, Germany.

[7] Kourtis A., et al. (2015). *Enhancing VNF Perfromance by exploiting SR-IOV and DPDK Packet Processing Acceleration*. IEEE Conference on NFV-SDN, USA.

[8] Liang C. et al. (2016). *Scalable High-Performance User Space Networking for Containers*. DPDK US Summit, San Jose, 2016.

[9] Tan, J. et al. (2107). *VIRTIO-USER: A New Versatile Channel for Kernel-Bypass Networks*. Association for Computing Machinery, USA.

[10] Russell, R. (2008). *virtio: Towards a De-Facto Standard For Virtual I/O Devices*. Association for Computing Machinery SIGOPS Operating System Review, USA.

[11] Berstein, B. (2014). *Containers and Cloud: From LXC to Docker to Kubernetes*. Column: Cloud Tidbits, IEEE Cloud Computing.

[12] McKeown, N. (2008). *OpenFlow: enabling innovation in campus networks*. Association for Computing Machinery SIGCOMM Computer Communication Review, USA.

[13] Joy, M. A. (2015) *Performance Comparison Between Linux Containers and Virtual Machines*. International Conference on Advances in Computer Engineering and Applications, IMS Engineering College, India.

[14] Khalid, J. et al. (2018) *Iron: Isolating Network-based CPU in Container Environments*. USENIX Symposium on Networked Systems Design and Implementation, USA.

[15] Szabo, M. et al. (2017) *Making the Data Plane Ready for NFV: An Effective Way of Handling Resources*. Association for Computing Machinery, USA.

[16] Suo, K. (2018). *An Analysis and Empirical Study of Container Networks*. IEEE Conference on Computer Communications, INFOCOM, USA.

[17] Intel, Corp. (2011). *PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology*. Revision 2.5. Available at https://www.intel.sg/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf. [Accessed: 13.04.2018].

[18] Intel, Corp. (2008). *Intel VMDq Technology: Notes on Softwre Design Support for Intel VMDq Technology*. Revision 1.2. Available at https://www.intel.com/content/www/us/en/virtualization/vmdq-technology-paper.html. [Accessed: 04.06.2018].

[19] Emmerich, P. et al. (2017). *Mind the Gap – A Comparison of Software Packet Generators*. Association of Computer Machinery, USA.

[20] Intel, Corp. (2014). *SR-IOV Configuration Guide*. Revision 1.0. Available at https://www.intel.com/content/dam/www/public/us/en/documents/technology-briefs/xl710-sr-iov-config-guide-gbe-linux-brief.pdf. [Accessed: 13.04.2018].

[21] Nangare, S. et al. (April 2018) *What is the Status of Containers and Microservices in Enterprises and Telecom*. Available at https://blog.calsoftinc.com/2018/04/status-of-containers-and-microservices-in-enterprises.html. [Accessed: 09.04.2018].

[22] Sayeed, A. (June 2016) *Is NFV ready for containers?* Available at https://www.redhat.com/blog/verticalindustries/is-nfv-ready-for-containers/ [Accessed: 22.04.2018].

[23] Natarajan, S. et al. (Noveber 2015) *An Analysis of Container-based Platforms for NFV*. IETF 94 Proceedings, Yokohama, Japan.

[24] DPDK Programmer's Guide. Available at http://doc.dpdk.org/guides-18.08/prog_guide/index.html [Accessed: 23.04.2018].

[25] DPDK Data Plane Development Kit. Available at https://www.dpdk.org/[Accessed: 2.2.2018].

[26] Linux Foundation Projects. Available at https://www.linuxfoundation.org/projects/. [Accessed: 23.04.2018].

[27] Free Software Foundation(2018). *Namespaces manual page*. Available at http://man7.org/linux/man-pages/man7/namespaces.7.html [Accessed: 10.10.2018].

[28] Free Software Foundation(2018). *Nsenter manual page*. Available at http://man7.org/linux/man-pages/man1/nsenter.1.html [Accessed: 10.10.2018].

[29] Free Software Foundation(2018). *Unshare manual page*. Available at http://man7.org/linux/man-pages/man1/unshare.1.html [Accessed: 10.10.2018].

[30] Free Software Foundation(2018). *Netlink manual page*. Available at http://man7.org/linux/man-pages/man7/netlink.7.htmll [Accessed: 10.10.2018].

[31] Cloud Native Computing Foundation(2018). *Container Network Interface Specification*. Available at https://github.com/containernetworking/cni/blob/master/SPEC.md [Accessed: 30.06.2018].

[32] Burns, B. et al, (2016) *Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade*. Association for Computing Machinery, vol. 14, issue 1.

[33] The Kubernetes Authors. *What is Kubernetes?*. Available at https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/. [Accessed: 30.06.2018].

[34] The Kubernetes Authors. *Creating Highly Available Clusters with kubeadm*. Available at https://kubernetes.io/docs/setup/independent/high-availability/. [Accessed: 30.06.2018].

[35] The Kubernetes Authors. *Concepts Underlying the Cloud Controller Manager*. Available at https://kubernetes.io/docs/concepts/architecture/cloud-controller/. [Accessed: 30.06.2018].

[36] Open vSwitch Documentation (2016) *Why Open vSwitch*. Available at https://github.com/openvswitch/ovs/blob/master/Documentation/intro/why-ovs.rst. [Accessed: 20.05.2018].

[37] Kubernetes Special Interest Groups. *Github project: Node Feature Discovery for Kubernetes*. Available at https://github.com/kubernetes-sigs/node-feature-discovery. [Accessed: 2.12.2018].

[38] Pfaff, B. et al, (2013) *The Open vSwitch Database Management Protocol*. Internet Engineering Task Force, Request For Comments. http://www.rfc-editor.org/rfc/rfc7047.txt

[39] The Kubernetes Authors. *Kubernetes Concepts: Annotations*. Available at https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/. [Accessed: 25.08.2018].

[40] The Kubernetes Authors. *Kubernetes Concepts: DaemonSet.* Available at https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/. [Accessed: 25.08.2018].

[41] Docker, Inc. (2016) (*Docker: Configure Networking.* Available at https://docs.docker.com/network/. [Accessed: 14.04.2018].

[42] Intel, Corp. *Github Project: SR-IOV CNI plug-in.* Available at https://github.com/intel/sriov-cni. [Accessed: 19.07.2018].

[43] Cerrato, I. et al, (2014) *Supporting Fine-Grained Network Functions through Intel DPDK.* European Workshop on Software-Defined Networks, Hungary.

[44] RedHat, Inc. *Github Project: Flannel.* Available at https://github.com/coreos/flannel. [Accessed: 20.07.2018].

[45] Open vSwitch Documentation. *Open vSwitch Manual: ovn-architecture - Open Virtual Network architecture.* Available at http://www.openvswitch.org/support/dist-docs/ovn-architecture.7.html. [Accessed: 01.11.2018].

[46] Vaughan, G. Duffy, D. Q.˙ *Containerizing the NASA Land Information System Framework.* Available at https://www.nas.nasa.gov/SC17/demos/demo29.html. [Accessed: 25.10.2018].

[47] Weaveworks. *Github Project: Flannel.* Available at https://github.com/weaveworks/weave. [Accessed: 20.07.2018].

[48] Calico Documentation. *Calico for Kubernetes.* Available at https://docs.projectcalico.org/v2.0/getting-started/kubernetes/. [Accessed: 20.07.2018].

[49] Open vSwitch Documentation. *OvS: Releases.* Available at http://docs.openvswitch.org/en/latest/faq/releases/. [Accessed: 15.07.2018].

[50] Bonafiglia, R., et al, (2015). *Assessing the Performance of Virtualization Technologies for NFV: a Preliminary Benchmarking.* European Workshop on Software Defined Networks, Spain.

[51] Mao, C., et al, (2015). *Minimizing Latency of Real-Time Container Cloud for Software Radio Access Networks.* IEEE International Conference on Cloud Computing Technology and Science, Canada.

[52] The Kubernetes Authors. *Master-Node Communication.* Available at https://kubernetes.io/docs/concepts/architecture/master-node-communication/. [Accessed: 07.09.2018].

[53] Canonical, Ltd. *Canonical Documentation.* Available at https://www.canonical.com/. [Accessed: 03.03.2018].

[54] Intel, Corp. *Github Project: Multus CNI plug-in.* Available at https://github.com/intel/multus-cni. [Accessed: 29.08.2018].

[55] Linux Foundation Collaborative Project. *How to use Open Virtual Networking with Kubernetes.* Available at https://github.com/openvswitch/ovn-kubernetes. [Accessed: 01.09.2018].

[56] Käldström, L. (2016). *Autoscaling a multi-platform Kubernetes cluster built with kubeadm.* KubeCon, Berlin.

[57] Open vSwithc Documentation. *Connecting VMs Using Tunnels (Userspace).* Available at http://docs.openvswitch.org/en/latest/howto/userspace-tunneling/. [Accessed: 11.08.2018].

[58] Long, T., Veitch, P. (2017). *A Low-Latency NFV Infrastructure for Performance-Critical Applications.* Intel Developer Programs.

[59] The Kubernetes Authors. *Managing Compute Resources for Containers: Resources.* Available at https://kubernetes.io/docs/concepts/configuration/manage-compute-resources-container/#resource-types. [Accessed: 22.09.2018].

[60] The Kubernetes Authors. *The Kubernetes Scheduler.* Available at https://github.com/fabric8io/kansible/blob/master/vendor/k8s.io/kubernetes/docs/devel/scheduler.md. [Accessed: 20.09.2018].

[61] Golang Documentation. *Core v1: Node.* Available at https://godoc.org/k8s.io/api/core/v1#Node. [Accessed: 20.09.2018].

A Appendix A

Code written in Golang to implement the basic Kubernetes scheduler along with a Pod manifest to choose this scheduler.

```
$ cat main.go
package main

import (
    "flag"
    "fmt"
    apiv1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/rest"
    "k8s.io/client-go/tools/clientcmd"
    "log"
    "os"
)

func main() {
    log.Println("Latency␣scheduler")

    // Parse arguments
    var SchedulerName string
    flag.StringVar(&SchedulerName, "scheduler-name", "latency
       -scheduler",
            "Name␣for␣the␣k8s␣scheduler")

    defaultKubeConfig := os.Getenv("HOME") + "/.kube/config"
    var ns, label, field, kubeconfig string
    flag.StringVar(&ns, "namespace", "default", "namespace")
    flag.StringVar(&label, "label", "", "Label␣selector")
    flag.StringVar(&field, "field", "", "Field␣selector")
    flag.StringVar(&kubeconfig, "kubeconfig",
       defaultKubeConfig,
            "Absoulte␣path␣to␣kubeconfig")
    var runInCluster = flag.Bool("run-in-cluster", false, "
       Run␣in␣cluster␣as␣a␣pod?")
    var timeout = flag.Int64("timeout", 5, "Timeout␣for␣
       watching␣for␣new␣pods")

    flag.Parse()

    // Get clientset. If running this process as a pod in the
        cluster,
    // remove both args in func BuildConfigFromFlags
    var config *rest.Config
```

```
var err error
if *runInCluster {
    config, err = rest.InClusterConfig()
} else {
    config, err = clientcmd.BuildConfigFromFlags("",
       kubeconfig)
}
if err != nil {
    log.Fatal(err)
}

clientset, err := kubernetes.NewForConfig(config)
if err != nil {
   log.Fatal(err)
}

corev1 := clientset.CoreV1()

listOptions := metav1.ListOptions{
    LabelSelector: label,
    FieldSelector: field,
}

nodes, err := corev1.Nodes().List(listOptions)
if err != nil {
    log.Fatal(err)
}

watchListOptions := metav1.ListOptions{
    Watch:          true,
    TimeoutSeconds: timeout,
    LabelSelector:  label,
    FieldSelector:  field,
}
podWatcher, err := corev1.Pods(ns).Watch(watchListOptions
   )
if err != nil {
    log.Fatal(err)
}

// go func() {
for {
    event := <-podWatcher.ResultChan()
    pod, ok := event.Object.(*apiv1.Pod)
    if !ok {
        log.Fatal("Failed␣retreiving␣pod")
    }
    if pod.Status.Phase == "Pending" && pod.Spec.
```

```
            SchedulerName == SchedulerName {
             SchedulePod(*pod, *nodes, clientset)
        }
}
    // }()
    fmt.Scanln()
    log.Println("Done")
}


$ cat scheduler.go
package main

import (
    "errors"
    apiv1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes"
    "log"
    "math/rand"
    "strconv"
)

const (
    LATENCY_WITH_OVS   = 5000   // With OvS (No overlay)
    LATENCY_WITH_MULTUS = 800   // With Multus
    LATENCY_DEFAULT     = 10000 //With OVN (Overlay)
)

func debugPrintNodes(comment string, nodes [](*apiv1.Node)) {
    log.Printf("%s: ", comment)
    for _, node := range nodes {
        log.Printf("%s\t", node.Name)
    }
    log.Printf("\n")
}

func Bind(pod *apiv1.Pod, node *apiv1.Node, clientset *
   kubernetes.Clientset) {
    log.Printf("Binding pod: %s to node: %s\n", (*pod).Name,
       (*node).Name)

    b := &apiv1.Binding{
        ObjectMeta: metav1.ObjectMeta{
                Namespace: (*pod).Namespace,
                Name:      (*pod).Name,
        },
        Target: apiv1.ObjectReference{
```

```
                Kind: "Node",
                Name: (*node).Name,
        },
    }
    corev1 := clientset.CoreV1()
    err := corev1.Pods((*pod).Namespace).Bind(b)
    if err != nil {
        log.Fatal(err)
    }
}

// Sort nodes to three groups - dpdk, ovs and default
// with priority in that order
func SortNodes(nodes apiv1.NodeList) (dpdk, ovs, def [](*
   apiv1.Node)) {
    var dpdkNodes, ovsNodes, defaultNodes [](*apiv1.Node)
    for _, node := range nodes.Items {
        for k, v := range node.Labels {
            if k == "dpdk" && v != "0" {
                dpdkNodes = append(dpdkNodes, &node)
                break
            } else if k == "ovs-dpdk" && v == "true" {
                ovsNodes = append(ovsNodes, &node)
                break
            } else {
                defaultNodes = append(defaultNodes, &node)
            }
        }
    }
    debugPrintNodes("DPDK␣Nodes", dpdkNodes)
    debugPrintNodes("OvS␣Nodes", ovsNodes)
    debugPrintNodes("Default␣Nodes", defaultNodes)
    return dpdkNodes, ovsNodes, defaultNodes
}

func FindNode(nodes apiv1.NodeList, requirement int) (*apiv1.
   Node, error) {
    log.Printf("FindNode␣with␣latency␣%d\n", requirement)

    dpdkNodes, ovsNodes, defaultNodes := SortNodes(nodes)
    switch requirement {
    case LATENCY_WITH_MULTUS:
        var found bool = false
        var numOfIfs int = 0
        var err error

        // If this node has some DPDK based interface, use
           this
```

```
        // and decrement the dpdk-interface count in node.
           Labels
        for _, node := range dpdkNodes {
                for k, v := range (*node).Labels {
                        if k == "dpdk" {
                            numOfIfs, err = strconv.Atoi(v)
                            if err != nil {
                                return nil, err
                            }
                            break
                        }
                }
                (*node).Labels["dpdk"] = strconv.Itoa(
                   numOfIfs - 1)
                // TODO update to api server
                found = true
                return node, nil
        }
        if !found {
                return nil, errors.New("No␣node␣meets␣
                   requirement")
        }

    // Pick a random node from the list for the next two
    //conditions
    case LATENCY_WITH_OVS:
        return ovsNodes[rand.Intn(len(ovsNodes))], nil

    case LATENCY_DEFAULT:
        return defaultNodes[rand.Intn(len(ovsNodes))], nil

    default:
        return nil, errors.New("Cannot␣be␣scheduled")
    }
    return nil, errors.New("Cannot␣be␣scheduled")
}

func FitPod(latency int, nodes apiv1.NodeList) (*apiv1.Node,
   error) {
    if latency < LATENCY_WITH_MULTUS {
        return nil, errors.New("Latency␣requirement␣cannot␣be
           ␣met")
    } else if latency > LATENCY_WITH_MULTUS && latency <
       LATENCY_WITH_OVS {
        return FindNode(nodes, LATENCY_WITH_MULTUS)
    } else if latency > LATENCY_WITH_OVS && latency <
       LATENCY_DEFAULT {
        return FindNode(nodes, LATENCY_WITH_OVS)
```

```
    } else if latency > LATENCY_DEFAULT {
        return FindNode(nodes, LATENCY_DEFAULT)
    }
    return nil, errors.New("Not a proper latency value")
}

func SchedulePod(pod apiv1.Pod, nodes apiv1.NodeList,
   clientset *kubernetes.Clientset) {
    log.Printf("Scheduling pod \"%s\" to nodes\n", pod.Name)
    for _, node := range nodes.Items {
        log.Printf("\t%s\n", node.Name)
    }

    var keyFound bool = false
    for key, val := range pod.Annotations {
        if key == "latency" {
            latencyVal, err := strconv.Atoi(val)
            if err != nil {
                log.Fatal(err)
            }

            node, err := FitPod(latencyVal, nodes)
            if err != nil {
                log.Fatal(err)
            }

            Bind(&pod, node, clientset)
            log.Println(val)
            keyFound = true
        }
    }
    if !keyFound {
        log.Println("Pod annotation with [latency: value] 
           needed")
    }
}

$ cat pod.yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox-latency
  annotations:
    latency : "100"
spec:  # specification of the pod's contents
  schedulerName: latency-scheduler
  containers:
  - name: busybox-latency-cont
```

```
    image: "busybox"
    command: ["top"]
    stdin: true
    tty: true
```

B Appendix B

The code written in Golang to check if a node has DPDK supported interfaces. This is run as a Daemon Set in Kubernetes.

```
$ cat main.go
package main

import (
        "flag"
        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
        "k8s.io/client-go/kubernetes"
        "k8s.io/client-go/rest"
        "k8s.io/client-go/tools/clientcmd"
        "log"
        "os"
)

func main() {
    var runInCluster = flag.Bool("run-in-cluster", false, "
       Run␣in␣cluster␣as␣a␣pod?")
    flag.Parse()

    var config *rest.Config
    var err error
    if *runInCluster {
        config, err = rest.InClusterConfig()
    } else {
        homeDir := os.Getenv("HOME")
        config, err = clientcmd.BuildConfigFromFlags("",
           homeDir+"/.kube/config")
    }
    if err != nil {
        log.Fatal(err)
    }

    clientset, err := kubernetes.NewForConfig(config)
    if err != nil {
        log.Fatal(err)
    }
```

```
    corev1 := clientset.CoreV1()

    hostname, err := os.Hostname()
    if err != nil {
        log.Fatal(err)
    }

    node, err := corev1.Nodes().Get(hostname, metav1.
       GetOptions{})
    if err != nil {
        log.Fatal(err)
    }

    // Get details of the node
    infoMap, err := GetInfo()
    if err != nil {
        log.Fatal(err)
    }

    if infoMap == nil {
        log.Println("No labels added to node")
        return
    }

    if node.Labels == nil {
        node.Labels = make(map[string]string)
    }
    for k, v := range infoMap {
        log.Printf("Adding node label %s: %s\n", k, v)
        node.Labels[k] = v
    }

    log.Println("Updating node info to Kubernetes API server"
       )
    corev1.Nodes().Update(node)
}


$ cat get-info.go
package main

import (
    "bufio"
    "os/exec"
    "strconv"
    "strings"
)
```

```
func CheckNameIntel(line string) bool {
    validIfNames := []string{"82571", "82572", "82573", "
       82574",
        "82583", "ICH8", "ICH9", "ICH10", "PCH", "PCH2", "
           I217",
        "I218", "I219",
        "82598", "82599", "X520", "X540", "X550",
        "82540", "82545", "82546",
        "82575", "82576", "82580", "I210", "I211", "I350", "
           I354", "DH89",
        "X710", "XL710", "X722", "XXV710",
        "FM10420"}

    for _, name := range validIfNames {
        if strings.Contains(line, name) {
            return true
        }
    }
    return false
}

// Returns a map, infoMap
// infoMap["dpdk"] = string(<no-of-dpdk-ifs>)
// infoMap["ovs-dpdk"] = "true"/"false"
func GetInfo() (map[string]string, error) {
    infoMap := make(map[string]string)

    // First look for DPDK interfaces
    lspciCmd := exec.Command("lspci")
    lspciOutput, err := lspciCmd.Output()
    if err != nil {
        return nil, err
    }

    var ifCount int = 0

    scanner := bufio.NewScanner(strings.NewReader(string(
       lspciOutput)))
    for scanner.Scan() {
        if strings.Contains(scanner.Text(), "Ethernet␣
           controller") &&           strings.Contains(
           scanner.Text(), "Intel") {
            if CheckNameIntel(scanner.Text()) {
                ifCount++
            }
        }
    }
    infoMap["dpdk"] = strconv.Itoa(ifCount)
```

```
    ovsCmd := exec.Command("ovs-vswitchd", "--version")
    ovsCmdOut, err := ovsCmd.Output()
    if err != nil {
        return nil, err
    }

    if strings.Contains(string(ovsCmdOut), "DPDK") {
        infoMap["ovs-dpdk"] = "true"
    } else {
        infoMap["ovs-dpdk"] = "false"
    }
    return infoMap, nil
}

$ cat Dockerfile
FROM golang:1.10.3

ADD . /go/src/github.com/alpha-black/k8s-node-iftype-discover
WORKDIR /go/src/github.com/alpha-black/k8s-node-iftype-
   discover
RUN go get "k8s.io/client-go/kubernetes"
RUN go get "k8s.io/client-go/rest"
RUN go install

$ cat ds.yaml
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: node-if-discovery
  namespace: kube-system
spec:
  selector:
    matchLabels:
      name: node-if-discovery
  template:
    metadata:
      labels:
        name: node-if-discovery
    spec:
      tolerations:
      - key: node-role.kubernetes.io/master
        effect: NoSchedule
      containers:
      - name: node-if-discovery-container
        image: k8s-node-iftype-ds:1.0
        command: ["./go/bin/k8s-node-iftype-discovery", "--
           logtostderr", "--run-in-cluster=true"]
```

```
        imagePullPolicy: IfNotPresent
```